Regularizing Discontinuities in Ray Traced Global Illumination for Differentiable Rendering

by

Peter Quinn

Department of Electrical and Computer Engineering

McGill University, Montréal, QC

April 2021

A thesis submitted to McGill University in partial fulfillment of the requirements of the degree of Master of Science (M.Sc.), Electrical Engineering.

Abstract

In recent years, neural network based machine learning (ML) models have demonstrated revolutionary performance in computer vision tasks, such as object recognition. These methods typically focus on only 2D images and lack an understanding of the 3D world that underlies the image. On the other hand, rendering, in the computer graphics field, is the process of creating a 2D image from a digital description of a 3D scene.

Viewing computer vision as an inverse rendering problem has led to a growing interest in differentiable rendering, the key idea being that perhaps by incorporating information about how 3D scenes result in 2D images, ML models that analyse 2D images could be improved. Differentiable renderers, like traditional renderers, generate images from digital descriptions of 3D scenes, but also allow for the computation of gradients for the output image with respect to the various input parameters in the scene. These input parameters can include the positions of objects, material properties, and camera pose. The gradients can be used in end-to-end training of machine learning models, in applications such as single-view 3D object reconstruction, or analysis-by-synthesis approaches for inverse graphics.

We introduce a novel differentiable path tracing algorithm where discontinuities in the rendering process are regularized through a blurring of the geometry. Our differentiable renderer implements full global illumination and has parameters for controlling the regularization, which allows for control over the smoothness of the loss landscape. Additionally, we also explore how differentiable renderers can be adapted to camera effects such as motion blur and depth of field. We successfully apply our system to solve several examples of challenging inverse rendering optimization problems that involve complex light transport scenarios.

Résumé

Récemment, les modèles d'apprentissage automatique basés sur les réseaux neuronaux ont démontré des performances révolutionnaires dans les tâches de vision par ordinateur telles que la reconnaissance d'objets. Ces méthodes se concentrent généralement uniquement sur les images 2D et ne comprennent pas le monde 3D qui sous-tend l'image. D'autre part, le rendu, dans le domaine de l'infographie, est le processus de création d'une image 2D à partir d'une description numérique d'une scène 3D.

Le fait de considérer la vision par ordinateur comme un problème de rendu inverse a conduit à un intérêt croissant pour le rendu différentiable, l'idée clé étant que peut-être en incorporant des informations sur la façon dont les scènes 3D donnent des images 2D les modèles d'apprentissage automatique qui analysent les images 2D pourraient être améliorés. Les systèmes de rendu différentiable, comme les systèmes de rendu traditionnels, génèrent des images à partir de descriptions numériques de scènes 3D, mais permettent également de calculer des gradients pour l'image par rapport aux paramètres définissant la scène. Ces paramètres d'entrée peuvent inclure les positions des objets, les propriétés des matériaux, et la pose de la caméra. Les gradients peuvent être utilisés dans l'entraînement de modèles d'apprentissage automatique, dans des applications telles que la reconstruction d'objets 3D à vue unique ('single-view') ou des approches d'analyse par synthèse pour le rendu inverse.

Nous introduisons un nouvel algorithme de traçage de chemin différentiable où les discontinuités dans le processus de rendu sont régularisées par un flou de la géométrie. Notre système de rendu différenciable implémente l'éclairage global et a des paramètres pour contrôler la régularisation, ce qui permet de contrôler la régularité du terrain des coûts. De plus, nous explorons comment les systèmes de rendu différenciable peuvent être adaptés aux effets de caméra tels que le flou de mouvement et la profondeur de champ. Nous appliquons avec succès notre système pour résoudre plusieurs exemples de problèmes d'optimisation du rendu inverse impliquant des scénarios complexes de transport de lumière.

Acknowledgments

First and foremost, a huge thank you to my supervisor Derek Nowrouzezahrai. Your insight and encouragement were essential in completing this work.

I would also like Cengiz Öztireli for taking the time to regularly share his experience, ideas, and advice for this project with me.

I want to thank Jérôme Parent-Lévesque for his work and his help during this project. This project would not have gotten started without you, nor would it have progressed as far as it did.

Thank you to my lab mates, Adrien, Damien, Jack, Joey, Sayantan, and Yang yang, as well as Krishna, for the many insightful discussions and advice during my time in the lab.

Thank you to my parents, Janice and Jim. Your endless support helped make all this possible. And last but not least thank you to my friends, who, just like for undergrad, helped keep me sane during this degree.

Contribution of Authors

All chapters in this thesis were authored by Peter Quinn.

Contents

1 Introduction **1**

1.1 Contributions 4

1.2 Thesis Overview 5

2 Background **6**

2.1 Rasterization 6

2.2 Ray Tracing 8

2.2.1 The Rendering Equation 9

2.2.2 Solving the Rendering Equation 13

2.3 Monte Carlo Integration 15

2.3.1 Monte Carlo Integration in Rendering 17

2.4 Caustics and Light Tracing 19

2.5 Camera Effects 22

3 Differentiable Rendering **28**

3.1 Discontinuities and Differentiability 28

3.2 Related Work 30

3.2.1 Rasterizers 31

3.2.2 Ray Tracers 33

3.3 Automatic Differentiation 38

3.4 Applications 40

4 Methodology **43**

4.1 Edge Discontinuity . . . 44
4.2 Occlusion Discontinuity . . . 46
4.3 Modifications to the Path Tracing Algorithm . . . 47
4.4 Explicit Connections and Shadows . . . 50
4.5 Material Model . . . 51
4.6 Textures . . . 51
4.7 Camera Effects . . . 52
4.7.1 Depth of Field . . . 52
4.7.2 Motion Blur . . . 52
4.8 Caustics and Light Tracing . . . 53

5 Results **55**

5.1 Forward Render Examples . . . 55
5.2 Optimizations . . . 56
5.2.1 Direct Lighting . . . 58
5.2.2 Higher-Order Light Transport . . . 59
5.2.3 Camera Effects . . . 60
5.2.4 Caustics . . . 62
5.3 Renderer Gradients . . . 64
5.4 Comparison with REDNER . . . 65

6 Discussion **69**

6.1 Benefits . . . 69
6.1.1 Implementation . . . 69
6.1.2 Camera Effects . . . 71
6.2 Limitations . . . 71
6.3 Differentiable Renders in General . . . 73
6.3.1 Exact Gradients and Bias . . . 73
6.3.2 Inclusion of Occlusion Discontinuity Differentiability . . . 74

6.3.3 Initialization 75
6.3.4 Self Intersections 75
6.4 Implicit Representations in Differentiable Rendering 76

7 Conclusion **79**

7.1 Differentiable Rendering – Perspectives and Future Work 80

List of Figures

1.1 Is this blue sphere shiny or painted? 2
1.2 Comparison of Rasterization and Ray Tracing 4
1.3 An rendered image with discontinuities explained 5

2.1 A high-level overview of the programmable rasterization pipeline 7
2.2 Real vs Render Image Comparison 8
2.3 Path tracing diagram 10
2.4 Direct and Indirect Illumination 11
2.5 Orientation of ω_i and ω_o around the shading point $\mathbf{x}$ 13
2.6 Reparameterization of incident light at point x in terms of outgoing radiance at another point found using the ray tracing function. 14
2.7 Next Event Estimation Diagram 19
2.8 Next event estimation geometry terms parameters 20
2.9 An example of a caustic 21
2.10 Light paths in a scene with a glass sphere 24
2.11 An example of depth of field 25
2.12 An example of motion blur 26
2.13 An example of camera effects in a Toy Story 4 27

3.1 Edge Discontinuities Diagram 29
3.2 Occlusion Discontinuities 30
3.3 Example from Soft Rasterizer 32
3.4 An example from DIB-R 33
3.5 Example from Redner 34
3.6 Example from Path Space Differentiable Rendering 35

3.7 Example from Mitsuba 2 . . . 36
3.8 An example from Radiative Back Propagation . . . 37
3.9 An example from Unbiased Warped Area Sampling . . . 38
3.10 Taxonomy of Differentiable Renderers . . . 39
3.11 Example application of Differentiable Rendering in Machine Learning . . . 41
3.12 Example of an Adversarial Classification . . . 42

4.1 Smoothed Edge Discontinuities . . . 44
4.2 Sampling Surfaces in Occlusion Discontinuities . . . 48

5.1 The blur around the edges of the triangles decreases as σ decreases. . . . 56
5.2 The blur over the depth of the objects decreases as γ decreases. . . . 56
5.3 Optimize the diffuse albedo of all primitive in the scene. . . . 58
5.4 Optimize the position of the camera. . . . 59
5.5 Optimize the position of the light. . . . 59
5.6 Optimize the position of an off screen object based on its shadow. . . . 60
5.7 Optimize the rotation of the cube (3 degrees of freedom). . . . 60
5.8 Optimize the texture on a surface that is only visible in a mirror-like surface. 61
5.9 Reconstructed Textures from Indirect Lighting . . . 61
5.10 Optimize the distance of the focal point of the camera to match the depth of field effect. . . . 62
5.11 Optimize to determine the vertices and velocity of the object in an image with significant motion blur. . . . 62
5.12 Snapshots from the motion blur geometry optimization, with no motion blur. Not used in the optimization process. . . . 63
5.13 Optimize the index of refraction of a sheet of glass with a normal map that focuses the light to produce a lens effect. . . . 64
5.14 Optimize the position of the caustic. . . . 64
5.15 Caustic Gradient . . . 65
5.16 Gradients with different values of σ . . . 66
5.17 Object Translation vs REDNER . . . 67

5.18 REDNER object translation with wider FOV 67

6.1 Example of Gaussian Blurs for Differentiable Visibility 70
6.2 An example from NeRF 78

List of Abbreviations

AD	Automatic Differentiation
BDPT	Bidirectional Path Tracing
BRDF	Bidirectional Reflectance Distribution Function
BSDF	Bidirectional Scattering Distribution Function
BVH	Bounding Volume Hierachy
FOV	Field of View
FPS	Frames per Second
GPU	Graphics Processing Unit
MC	Monte Carlo
ML	Machine Learning
MLP	Multi Layer Perceptron
NN	Neural Network
PDF	Probability Density Function
RGB	Red-Green-Blue, the 3 colour channels in a pixel
RMS/RMSE	Root Mean Square Error
SGD	Stochastic Gradient Descent
SPP	Samples per Pixel

Chapter 1

Introduction

In the past decade, the field of computer vision has seen significant advancements due to the proliferation of machine learning (ML) techniques and specialized neural network architectures [1, 2]. However, many of these modern techniques focus only on the 2D image and are agnostic to the underlying 3D world and geometry that are behind the image formation process. When we humans look at an image, our understanding and experience of the 3D world allow us to infer more general information about the scene in the image. For example, if we see a bright spot on a glossy surface, we reason that it is due to a reflection of a light source, rather than a small part of the surface being painted in such a way that it appears brighter, such as in Figure 1.1.

Inverse graphics is the process of obtaining a detailed description of a 3D scene from an image. One technique that is useful in inverse graphics is differentiable rendering, which allows for the computation of gradients through the rendering process [3, 4]. These gradients can then be used to optimize estimates of scene parameters to closely match a target image.

Recently, differentiable rendering has been increasingly used in different ML applications such as generating a 3D representation of an object from a 2D image (mesh reconstruction) [5] and estimating the orientation and intensity of light sources from an image

Figure 1.1: Are the white areas on this blue sphere due to the sphere being glossy and reflecting the light from a nearby light source, or could the sphere be painted in such a way that it gives this illusion? How could you tell? *Source:* `https://image.freepik.com/free-vector/realistic-blue-sphere-with-shadow_6735-671.jpg`

(lighting reconstruction) [6]. The key idea is that incorporating an explicit process of converting 3D geometry into 2D images may allow for machine learning models to better generalize information from images.

Differentiable rendering involves modifying the rendering process to remove discontinuities in the computation of pixel colour, which would otherwise prevent useful gradients from being calculated. There are two main methods for rendering images, rasterization and ray tracing, both of which have been explored for differentiable rendering.

Rasterization uses multiple matrix multiplications to efficiently project the geometry in the scene onto the plane defined by the camera position, direction and field of view. This comes at the cost of realism since light paths consisting of multiple bounces are ignored. Several works such as SOFTRAS [5] and DIB-R [6], and NEURAL MESH RENDERER [7] have implemented rasterization-based differentiable renderers and applied them to ML applications.

On the other hand, ray tracing produces a detailed, physically accurate image by simulating light rays propagating through a scene. Images produced using ray tracing can capture complex lighting effects accurately, at the cost of more computation time. Some differentiable rendering techniques for ray tracing have appeared in the graphics literature in recent years [8, 9, 10, 11, 12, 13], but their high computational cost has limited their use in ML applications.

While the rendering process is easily differentiable with respect to the material properties at a point on the surface of an object (given a reasonably simple material model), getting gradients with respect to the scene geometry requires more work. In traditional rendering algorithms, there are essentially two discrete, non-differentiable steps that need to be modified to make the rendering differentiable with respect to geometry: Discontinuities at the edges of geometry, and determination of the surface closest to the camera.

Differentiable rendering is attractive in an ML setting as it presents an efficient way of doing unsupervised image-based ML applications. In these applications, we desire to train a model to extract some kind of 3D information (such as shape, object position, or colour) from a target image without having a reference as to what the correct 3D information is. Normally, this would make training a model very difficult, but by using a differentiable renderer we can render a 2D image based on the extracted 3D information, and compare this 2D image to the input image. By comparing the rendered image and the input image, we can calculate a loss. As the rendering process was differentiable, we can backpropagate gradients through it to update the parameters of our ML model.

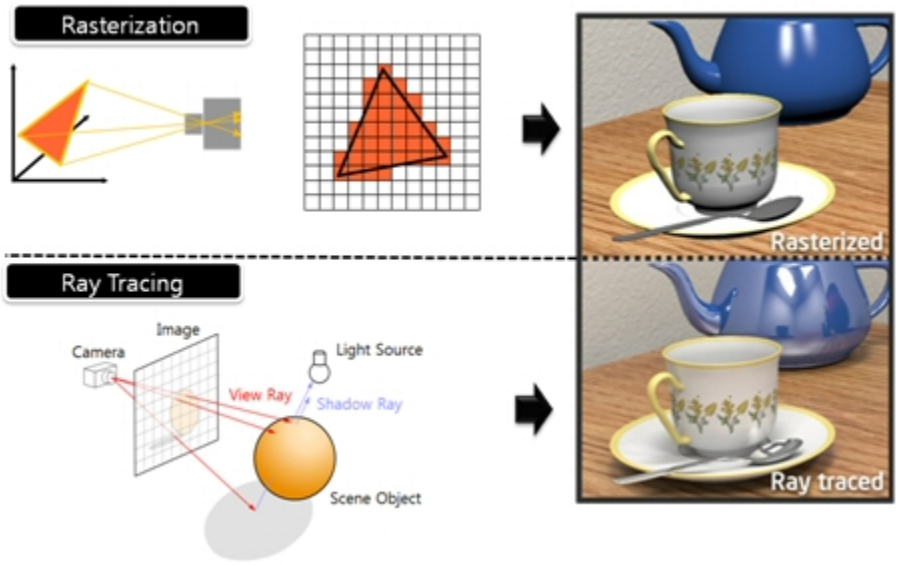


Figure 1.2: A comparison of rasterization and ray tracing. With rasterization, the triangles that form the objects are projected onto the 2D surface that forms the image. Based on this 2D projection, each pixel is filled in with the appropriate shading based on the closest surface. With ray tracing, we cast many rays from the camera through each pixel and simulate their interactions in the 3D scene. This leads to a more realistic image, which can be noticed in the shadows around the plate edges and the reflections in the teapot. *Source: http://mvlsi.kaist.ac.kr/research/multimedia-processor/ray-tracing*

1.1 Contributions

We present a manuscript for a novel formulation for differentiable path tracing based on an extension of the probabilistic view of triangle visibility presented by [5] that model's global illumination effects while allowing gradient computation with respect to both geometry and material properties. We implement this formulation in Python using the PyTorch library to allow for simple integration with other common machine learning libraries and pipelines. We apply our technique to several example inverse rendering problems involving complex light transport effects such as shadows, indirect lighting, reflections, and optimization of object and vertex positions. Additionally, we show results

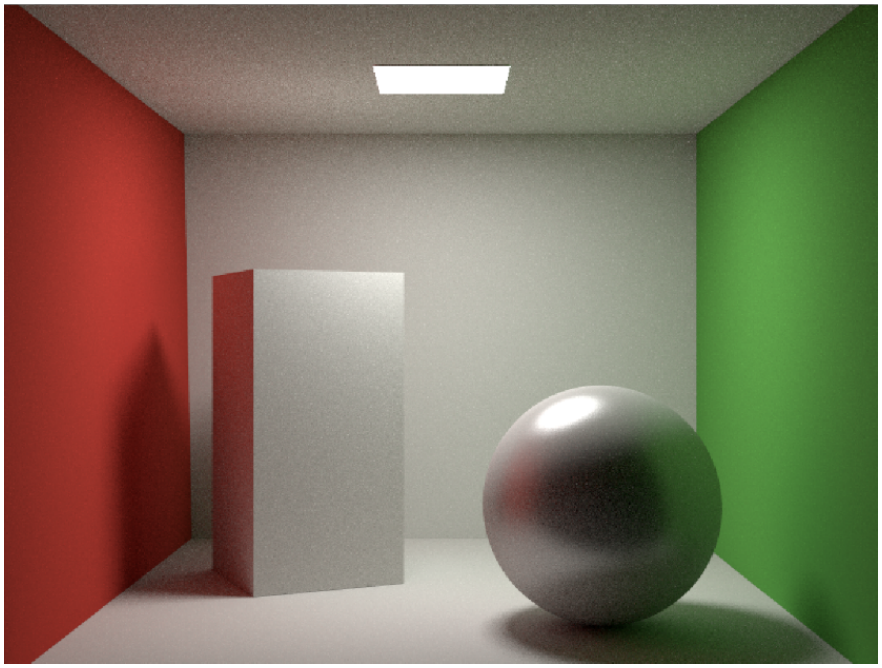


Figure 1.3: The edges of geometry in this image are discontinuities. Moving from the red wall on the left to the back wall, there is a sudden change in the value of adjacent pixels, as we cross the edge and go from red to white. Additionally, there are discontinuities where the box and sphere occlude the back wall, as the rendering process determines which object appears in front of the other, discarding information about the surfaces that are occluded.

on optimizing for camera distribution effects such as depth of field and motion blur. To the best of our knowledge, our method is the first to support such effects.

1.2 Thesis Overview

Chapter 2 describes how images are generated using computers. Chapter 3 reviews differentiable rendering and the recent literature in that area. Chapter 4 describes our novel differentiable rendering method. Chapter 5 shows a variety of examples and comparisons that demonstrate the capabilities of our differentiable renderer. Chapter 6 offers an extended general discussion of our results and differentiable rendering. Finally, Chapter 7 concludes the thesis and addresses possible avenues of future work.

Chapter 2

Background

There are two main approaches to project 3D geometry onto 2D images are rasterization and ray tracing. The original work in this thesis is focused on ray tracing, but rasterization is briefly summarized here for completeness. Ray tracing is reviewed in more detail, with the relevant mathematical foundations and equations being presented.

2.1 Rasterization

Rasterization is a method that relies on projecting an image of a 3D scene onto the 2D viewing image. This projection is done by applying a series of matrix transformations to the objects in the scene, to rotate, translate, scale, and apply perspective appropriate to the viewing angle. This series of matrix transformations can become very computationally expensive when there are a large number of objects and when the meshes describing the objects are very detailed, with many vertices to handle.

To handle detailed scenes, computers will typically use a graphics processing unit (GPU), which is a piece of hardware optimized for performing the necessary matrix operations. GPUs are typically able to render complex scenes (e.g. scenes with millions of

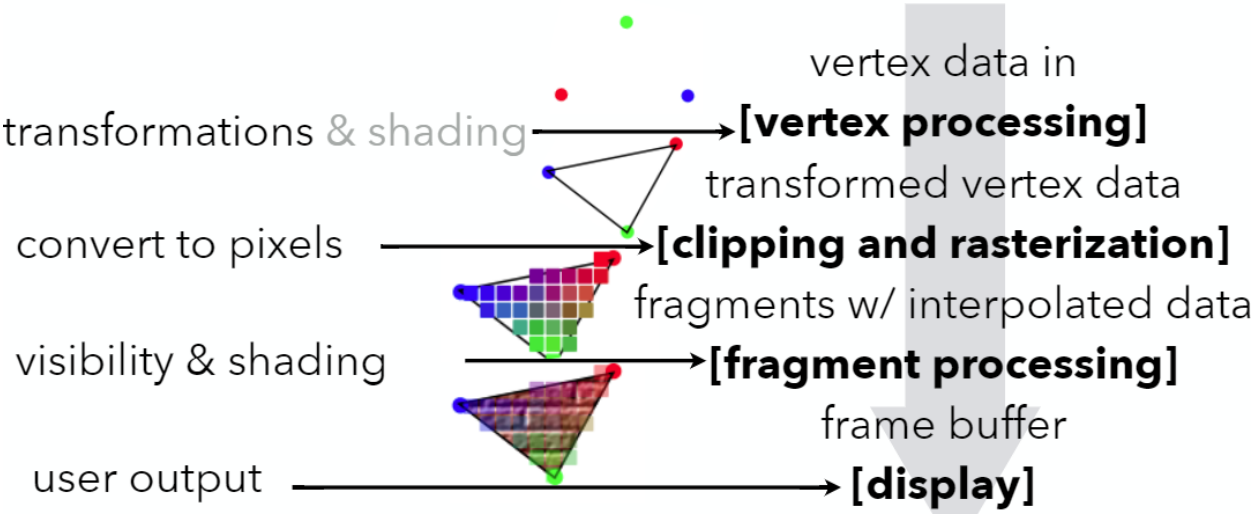


Figure 2.1: A high-level overview of the programmable rasterization pipeline that exists on modern GPUs. First, the vertices of the input objects are scaled, translated and rotated as desired. These transformations are used to modify the shapes of objects and then determine what point on the 2D image plane the 3D object point projects to. Second, the areas in between the vertices on the 2D plane are filled in during *rasterization*. Pieces of triangles that do not lie in the image plane are discarded in a process known as *clipping*. We also determine which triangle on the image plane is the closest to the camera, and therefore visible, by keeping track of the distance to the closest triangle for each pixel in a z-buffer. These remaining pieces of the objects are known as fragments. Finally, the fragments are passed to a fragment shader, which determines their final appearance, and the image is stored in a frame buffer before being output to the display. *Source: ECSE 546 Course Notes, Derek Nowrouzezahrai*

triangles) at interactive rates (i.e 60 Hz). This ability to render images at this rate makes it very attractive for use in video games and 3D modelling software.

There are significant limitations to rasterization. Rasterization does not consider how light travels and bounces through a scene, so effects such as reflections and indirect lighting do not appear, such as in Figure 1.2. Even accurate shadows are very difficult to produce with rasterization and are typically only approximated using techniques such as shadow mapping.

These higher-order light transport effects are often critical to achieving realistic-looking computer-generated images. Fortunately, ray tracing is designed to accurately handle these situations. Recent advances in GPU hardware have added specialized ray tracing cores to GPUs (such as in the Nvidia RTX 2060), to supplement the rasterization pipeline with additional graphical fidelity from hardware-accelerated ray tracing.

2.2 Ray Tracing

Ray tracing the preferred technique for creating high-quality images. This technique can render photorealistic images with a high degree of accuracy, including effects such as soft shadows from area or multiple light sources, indirect illumination, reflections, refraction, and sub-surface scattering. Often, these images can be such high quality that they are indistinguishable from photographs, such as in Figure 2.2.

(a) **(b)**

Figure 2.2: One of these is a real photograph, one is rendered. Which is which?[1]

Ray tracing involves simulating many rays of light, determining how they interact with the objects in the scene based on objects' material properties, and calculating how this appears to the camera based on the camera's position and properties. A scene is described by some digital file detailing the shape, position, colour, texture, etc., of the objects and light sources. A simple example is shown in Figure 2.3

In general, ray tracing can be considered as any algorithm that generates rays and intersects them with the scene geometry for computing physically-based illumination. In this section, we will focus our discussion on algorithms that use ray tracing to solve the "rendering equation" [14] to compute global illumination, relying on Monte Carlo integration.

There is one major downside to ray tracing. It can take significant time to create detailed, high-quality images. The time taken can span from a few seconds to several minutes or hours, for a single image or frame of video. While a few seconds might not seem like much, this is much too slow for use in video games or other interactive media. It takes at least 24 frames per second (FPS) for video motion to seem smooth to the human eye. This means we have at most 40ms to render an image to produce smooth video. In practice, we usually want 30 or even 60 FPS to ensure the movement appears convincingly smooth and responsive to inputs in the case of video games. This further reduces our max computation time for a frame to 32ms or 16ms or even less.

The long rendering time for ray tracing is due to the computational complexity of simulating millions of light rays travelling and bouncing through a 3D scene. As such, this technique has been restricted to applications where images and videos can be rendered ahead of time, then simply played back. As such, it is commonly used for computer-generated images (CGI) in TV and film, as well as pre-rendered cut-scenes in video games.

2.2.1 The Rendering Equation

Most often when rendering scenes, we are interested in computing the *global illumination* at the points we can see from our viewing position. Global illumination refers to the illumination due to both the light coming directly from light sources (direct illumination) and the indirect light due to reflections off of other objects in the scene (indirect illumina-

[1] **(a)** is real; **(b)** is rendered.
Images source: https://www.azuremagazine.com/article/spot-the-fake-real-or-rendering/

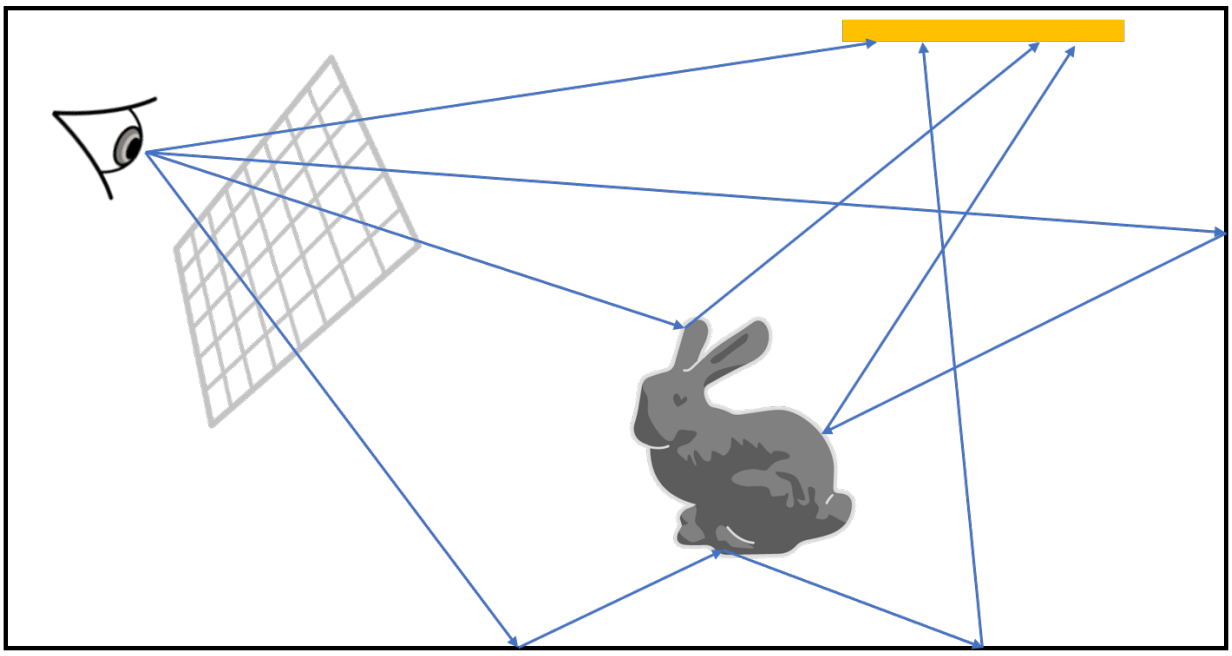


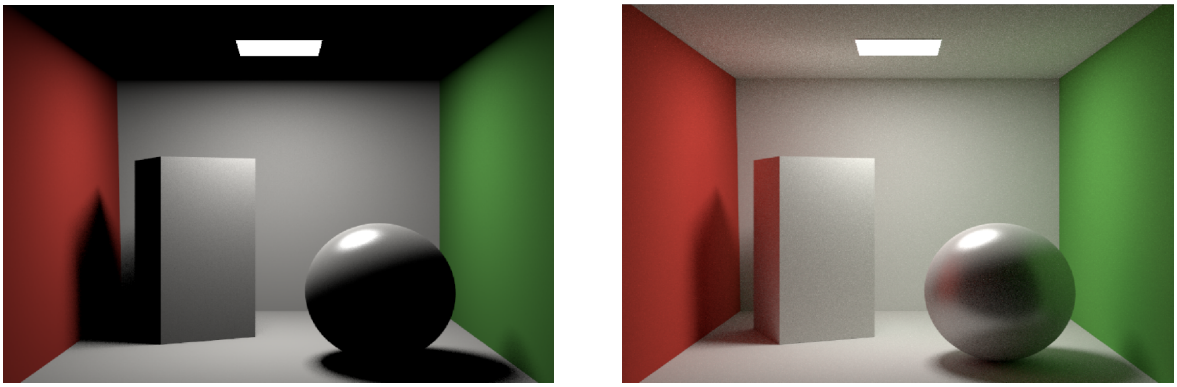
Figure 2.3: Starting from the eye, we generate rays that pass through the pixel grid and simulate their interactions with the 3D scene, allowing them to bounce around until they connect to a light source. Simulating many of these light paths and averaging their contributions allow us to generate a detailed, photorealistic image. As there are an infinite number of possible ways to connect the eye and the light sources, we must simulate a large number of paths and use numerical methods to ensure our estimates converge.

tion). An example of the contributions of direct and indirect lighting can be seen in Figure 2.4.

A compact equation that describes the global illumination at any point in the scene is given by the Rendering Equation [14]. After being proposed by Jim Kajiya in 1986, most research into photorealistic rendering has focused on solving this equation as quickly and as accurately as possible.

The equation is shown here in Equation 2.1

$$L_o(\mathbf{x}, \omega_o) = L_e(\mathbf{x}, \omega_o) + \int_{H^2} f_r(\mathbf{x}, \omega_i, \omega_o) L_i(\mathbf{x}, \omega_i) \cos\theta_i d\omega_i \tag{2.1}$$



(a) **(b)**

Figure 2.4: (a) Direct illumination in simple scene. Note the lack of illumination on the ceiling and the left side of the rectangular prism. These areas do not directly view the light, and as a result, appear black. **(b)** Global illumination in a simple scene. Global illumination is the sum of the direct and indirect illumination components. Indirect illumination was capped at four light bounces.

This equation shows that the radiance L_o (Units: $\mathrm{W} \cdot \mathrm{sr}^{-1} \cdot \mathrm{m}^{-2}$) leaving the point $\mathbf{x}$ in direction ω_o is equal to the sum of the radiance emitted by the point L_e and an integral term that represents the reflected radiance, often abbreviated as L_r.

The emitted radiance L_e is dependent on the point of interest $\mathbf{x}$ and the outgoing direction ω_o. This term is only non-zero if the point $\mathbf{x}$ is on the surface of a light source, typically called an emitter in rendering. Some light sources can emit more strongly in certain directions than other directions, giving this term a directional dependence.

The integral computes the total amount of light reflected at the point $\mathbf{x}$ in the outgoing direction ω_o, often shortened to L_r. This integral has three terms in it.

This first term f_r is the Bidirectional Reflectance Distribution Function (BRDF). It is sometimes more generally called a Bidirectional Scattering Distribution Function (BSDF). This term is the ratio of how much incoming light from direction ω_i gets reflected in the outgoing direction ω_o at $\mathbf{x}$.

The second term L_i is the radiance incoming to $\mathbf{x}$ from direction ω_i. The incident radiance could be originating directly from a light source in the scene, or it could correspond to light that has reflected off another surface to arrive at this point. This incoming radiance will be scattered (and possibly absorbed) by the material the forms the surface $\mathbf{x}$ lies on.

The third term, $\cos \omega_i$ is called the cosine foreshortening term. This term represents how the incoming energy gets spread out (in a differential sense) for a ray whose direction has an angle of θ_i with respect to the normal at the point $\mathbf{x}$. For a ray that is coming straight at a point, i.e. from an angle of 0, the energy is not spread out at all, and this term evaluates to 1. But as the ray comes from an increasingly grazing angle, the contribution of radiance to the point is reduced, eventually becoming 0 when the ray forms a right angle with the normal.

We integrate these three terms over the hemisphere, denoted by the shorthand $\int_{H^2}$. This integration effectively adds up the contribution of incoming lighting from every possible direction visible from the $\mathbf{x}$.

Physically based BRDFs

For a material to be physically based (i.e. obey the laws of physics, as we currently understand them), the BRDF must obey two constraints: Helmholz reciprocity and conservation of energy.

Helmholz Reciprocity states that the function should be symmetrical with respect to the going and outgoing directions. Mathematically, this is given in Equation 2.2.

$$f_r\left(\mathbf{x}, \omega_i, \omega_o\right) = f_r\left(\mathbf{x}, \omega_o, \omega_i\right) \tag{2.2}$$

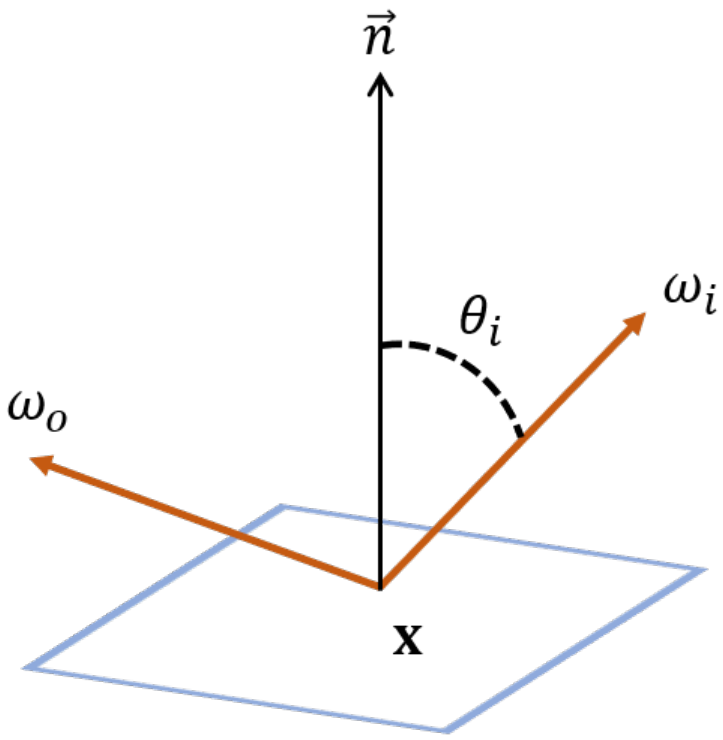


Figure 2.5: A diagram showing the relationship between the point $\mathbf{x}$, and the directions ω_i and ω_o. By convention, both ω_i and ω_o are unit vectors that point away from the shading point $\mathbf{x}$. The angle θ is the angle between the surface normal and ω_i.

Conservation of energy ensures that it it not possible for a point to reflect more energy than is incoming to it. This is given in Eq. 2.3. We allow for a point to absorb energy, which can result in the point reflecting less energy than is incident to it:

$$\int_{H^2} f_r\left(\mathbf{x}, \omega_i, \omega_o\right) d\omega_i \leq 1, \forall \omega_o \tag{2.3}$$

.

2.2.2 Solving the Rendering Equation

In practical implementations, global illumination must be calculated for each of the three colour channels (RGB). The red, green and blue components of the BRDF at any point

will depend upon the colour of the surface the point lies on. The incident radiance at point x from direction ω_i is equivalent to the outgoing radiance from another point in the scene. This point will lie at the intersection point of the ray originating at x in direction ω_i and the first surface it intersects with in the scene. Using this, we can reformulate the rendering equation into the following form:

$$L_o\left(\mathbf{x}, \omega_o\right) = L_e\left(\mathbf{x}, \omega_o\right) + \int_{H^2} f_r\left(\mathbf{x}, \omega_i, \omega_o\right) L_o\left(r\left(\mathbf{x}, \omega_i\right), -\omega_i\right) \cos\theta_i d\omega_i \tag{2.4}$$

$r(x, \omega_i)$ is the ray tracing function which returns the point of intersection between the ray traced in direction ω_i from point x and the first surface that this ray intersects. A diagram demonstrating this reparameterization can be seen in Figure 4.

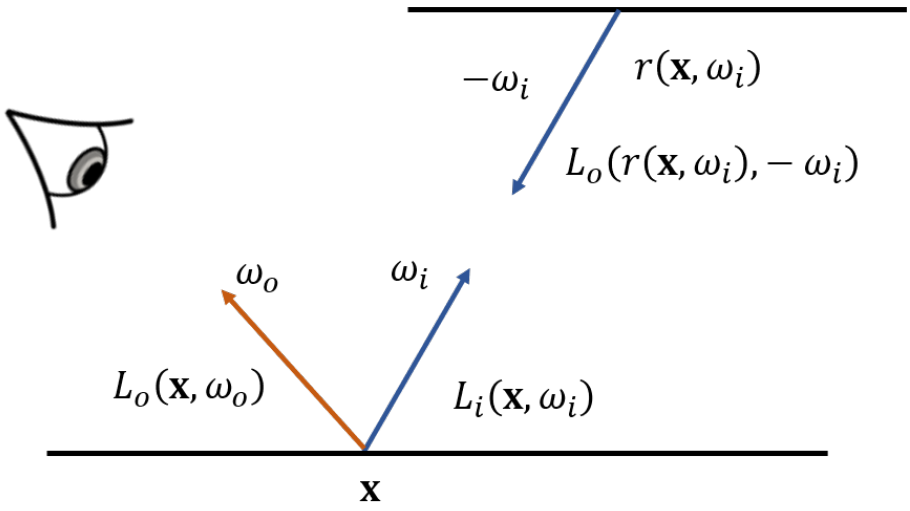


Figure 2.6: Reparameterization of incident light at point x in terms of outgoing radiance at another point found using the ray tracing function.

We can see that this equation now takes an interesting form. The dependence on the incident radiance $L_i(x, \omega_i)$ has disappeared and the outgoing radiance $L_o(x, \omega_o)$ now appears on both sides, where it appears inside the integral on the right-hand side. This makes the function recursive. This type of equation is known as a Fredholm equation of the second kind.

Solving the rendering equation is the primary challenge in rendering realistic scenes. Several algorithms exist that try to solve the rendering equation efficiently. This work focuses on the path tracing algorithm. This algorithm relies on Monte Carlo (MC) methods to numerically approximate the integral term.

2.3 Monte Carlo Integration

Monte Carlo (MC) integration is a numerical method for calculating a definite integral using random numbers. The name for the technique is a reference to the city of the same name in Monaco, which is famous for its random number generators, more commonly called casinos.

While MC integration was initially used to simulate particle interactions in nuclear physics [15], it is now frequently used in computer graphics to evaluate integrals, such as the rendering equation. A brief review of the fundamentals of Monte Carlo integration is provided here.

Given some function $f(x)$ and a random variable X with probability distribution $p(x)$, and $p(x_i) > 0, \forall f(x_i) \neq 0$ for all $x_i \sim X$ the following holds:

$$\int_\Omega f(x)du = \int_\Omega \frac{f(x)}{p(x)} p(x)du = E\left[\frac{f(X)}{p(X)}\right] \tag{2.5}$$

The expectation can be approximated using a finite number of samples N drawn from $p(x)$ using the following MC estimator:

$$\int_\Omega f(x)du \approx \frac{1}{N}\sum_{i=1}^{N} \frac{f\left(x_i\right)}{p\left(x_i\right)} \tag{2.6}$$

Essentially, the integral is approximated as the average of the integrand $f(x)$ weighted by $p(x)$ for N samples drawn according to the probability distribution $p(x)$. In the limit where $N \to \infty$, a strict equality holds.

MC integration is an attractive technique because the method requires only three conditions to compute unbiased estimates for an integral:

1. a probability distribution function $p(x)$ that is non-zero everywhere $f(x)$ is non-zero,
2. the ability to draw random samples according to $p(x)$, and
3. the ability to evaluate the integrand $f(x)$ at x

Monte Carlo integration provides an unbiased estimator for an integral. This means that there is no offset between the mean of our estimator and the integral we are trying to evaluate. The Monte Carlo method is easily extendable to higher dimensions; our scalar x simply becomes a vector $\bar{\mathbf{x}}$. Monte Carlo integration is very useful in evaluating higher-dimensional integrals because it does not become significantly more complex with additional dimensions.

Monte Carlo integration has two main drawbacks. The first issue is that there is "noise" in our image. This noise manifests in the image as grainy, uneven shading of areas that should appear uniform. This noise is caused by the variance due to the random sampling. Our MC estimator only converges perfectly if we do an infinite number of samples. For any finite number of samples, the variance will cause the estimate to differ from the expected value by some amount. As we do more samples, the variance of the estimator and the visual impact of the noise in the image decrease. This leads us directly to the second issue with MC integration: slow convergence rate.

MC integrals converge at a rate of $\mathcal{O}(\frac{1}{\sqrt{N}})$. This means that if we want to improve the quality of the image by a factor of 2, we need to take 4 times as many samples, and if we want to improve by a factor of 10, we need to take 100 times as many samples.

2.3.1 Monte Carlo Integration in Rendering

In the rendering equation, it is possible to satisfy all three conditions necessary for Monte Carlo. Our MC estimator for the integral in Equation 2.1 will be:

$$\int_{H^2} f_r(\mathbf{x}, \omega_i, \omega_o) L_i(\mathbf{x}, \omega_i) \cos\theta_i d\omega_i \approx \frac{1}{N} \sum_{j=1}^{N} \frac{f_r(\mathbf{x}, \omega_i^j, \omega_o) L_i(\mathbf{x}, \omega_i^j) \cos\theta_i^j}{p(\omega_i^j)} \tag{2.7}$$

Unfortunately, attempting to solve this estimator completely for a significant number of samples, while allowing for several bounces of indirect light at each point, is not practical. The recursion in the rendering equation results in an exponential growth of the paths needed to trace to compute the radiance at a point, which would require an impractical amount of computer memory and compute time.

This limitation leads us directly to the Path Tracing algorithm. Instead of taking N samples at a point, we will instead take a single sample and repeat this for however many bounces of indirect light we would like our light transport simulation to take into account. If a sufficient number of paths for each pixel in the image are traced and have their contributions averaged, the result will converge to the correct image.

We can actually do even better than this. If the path length is fixed to maximum depth, bias will be introduced into the estimation, as some amount of the light travelling around in the scene will be discarded. In practice, this bias is very small if a sufficiently high number of bounces is chosen, and will an imperceptible effect on the final image. This bias can be removed entirely by employing a technique called Russian roulette which stochastically determines when to terminate the path with some probability p, and reweights radiance contributions of path segments according to p. This results in a completely unbiased image.

In our renderer, we focus mainly on forward path tracing with explicit connections to light sources. This means that we start tracing rays from the camera. This is referred to as forward since this is how ray traced images are usually formed.

Having the rays originate from the camera has several advantages, mainly we ensure that all light paths we simulate will be visible to the camera, and we can create paths in such a way that ensures that all pixels in our image will have some number of samples taken.

Explicit connections to light sources (sometimes called *next event estimation*) is a technique to reduce the variance and improve the convergence rate of the image. Each time we get to a new point along the light path, before randomly choosing a new direction to continue the path in, we make a direct connection to a light source, by sampling a random point on its surface, as illustrated in Figure 2.7. We can do this because we have access to all the information about all the light sources in the scene, including their shape and position. By including an additional direct connection to a light source, we no longer have to rely on randomly hitting a light source with our trace ray (which is known as implicit path tracing).

We have to include an additional term when evaluating the integrand for this connection to ensure that our probabilities are correct. This is because we sampled a point over the area of the light source, rather than a random direction, so we need to include a Jacobian factor that corrects for the change in measure between directions and areas. This factor is often called the *geometry term* and is denoted by $G(x \leftrightarrow y)$. This factor is proportional to the inverse square of the distance between the points, and the cosines of the connection direction and normals of the surfaces, as shown in Figure 2.8.

$$G(x \leftrightarrow y) = \frac{\cos(\theta_x)\cos(\theta_y)}{||x - y||^2} \tag{2.8}$$

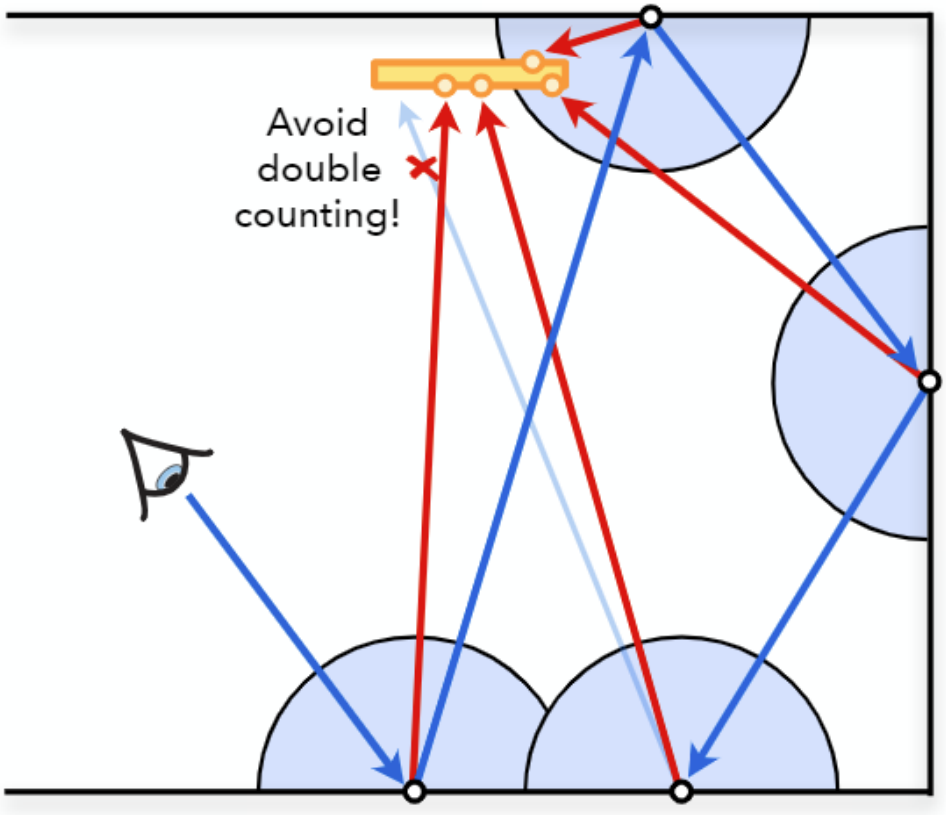


Figure 2.7: We can make our light transport simulation converge more quickly by using next event estimation. When we intersect a diffuse surface, we can sample a point on a light source in the scene and check if it's visible from the intersection point (red arrows). We can compute the contribution from this path before sampling a direction to continue bouncing (blue arrows). This breaks the integral computation into two parts, one for direct lighting and the other for indirect lighting. We must ensure that our samples intended for estimating indirect lighting do not intersect emitters, otherwise, we would double count direct lighting contributions. *Source: ECSE 546 Course Notes, Derek Nowrouzezahrai*

2.4 Caustics and Light Tracing

When light passes from one medium to another medium with a different index of refraction, it undergoes a phenomenon known as refraction. Refraction causes the path of the

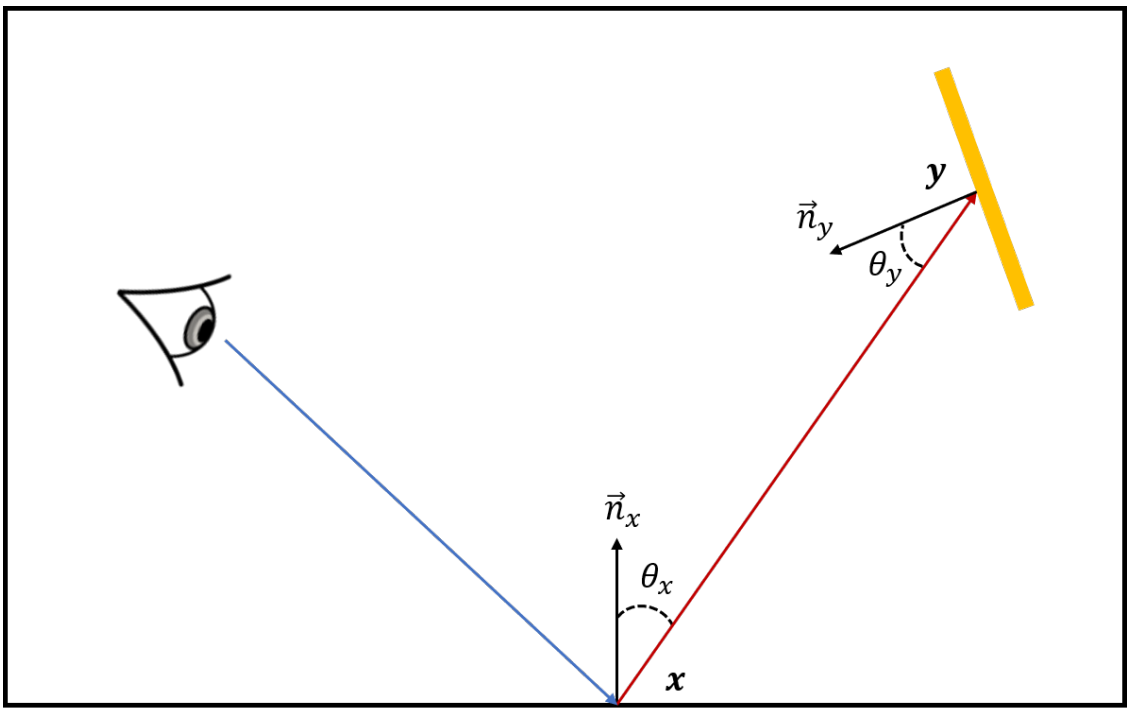


Figure 2.8: The relationships between the terms used in the computation of the geometry term $G(x \leftrightarrow y)$ when using next event estimation. Usually, a separate check is done to ensure that the normal of the intersection point and the normal of the sampled point on the emitter are aligned correctly.

light to bend proportionally to the ratio of the indices of refraction of the two materials. The exact change in direction can be calculated using Snell's Law [16].

When travelling from one material into a different material with a higher index of refraction (such as from air into glass), the bending of light paths causes the energy of the incoming light paths to be focused into a smaller range of angles [17]. Visually, this manifests as a patch of brightness, typically referred to as a caustic.

When ray tracing, producing an image with caustics takes special care. The interactions at the boundaries between materials are specular, which means that for an incoming light path, there is a unique outgoing direction. This means that when ray tracing with ray originating from the camera, we cannot explicitly form connections to a light source from a point on the material boundary. This explicit connection assumes that there is a diffuse scattering interaction occurring at the point, which is not the case in this situation.

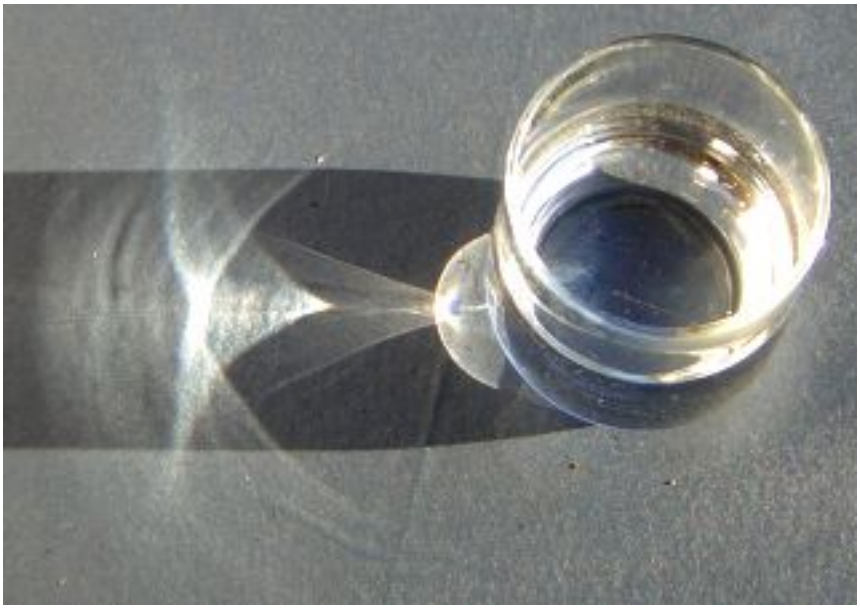


Figure 2.9: A caustic cast from a glass of water with sunlight incident from the right side of the image. The light passing through the water and glass is refracted and focused, resulting in the areas of brightness. *Source: Heiner Otterstedt, https://commons.wikimedia.org/w/index.php?curid=5900818*

One possible solution is to fall back on implicit path tracing, relying on random chance to find the light paths that connect to a light source and result in caustics. In practice, this method is terribly inefficient, taking a long time to converge, and only works for light sources with finite area. Generating a caustic from a point light would be mathematically impossible using our random sampling techniques, as we would have to randomly hit an infinitely small point in space, which has probability 0. A more efficient method is to invert our typical method for ray tracing and to trace rays beginning from the light sources. This technique is typically called light tracing.

Practically, this technique ends up being very similar to the forward path tracing discussed previously. We sample a point and direction from the light sources in our scenes, ray trace and resolve the intersections, then we can form an explicit connection to our camera when we encounter a diffuse surface. This will allow us to efficiently find paths that result in caustics.

Light tracing also has its limitations. When a point/pinhole camera with light tracing, it is actually impossible (i.e. it has a probability of 0) to find paths that have a specular interaction that connects to the camera through random sampling. The reason for this is identical to how we cannot find caustics for point light sources with path tracing. This means that when we try to render a mirror or a piece of glass with light tracing, they will appear completely black.

Ideally, we would combine the strengths of both path tracing and light tracing to overcome the limitations of each individual one. Fortunately, this is exactly what the technique of bidirectional path tracing (BDPT) does [18]. Unfortunately, even with BDPT, there are still some light paths that are difficult to form, specifically ones that have a diffuse interaction in between two specular interactions. These are sometimes referred to as *SDS* (specular-diffuse-specular) paths. For these, methods such as photon mapping exist [19] [20].

2.5 Camera Effects

When rendering an image, everything describing the is done digitally; the shape and colour of the objects, the lighting, as well as the camera used to ”take” the picture. Real cameras have various physical constraints and properties that affect how they form an image. When describing a scene, we have to decide if and how we want to replicate these effects.

The most common camera model used in rendering is the pinhole camera. This is an idealized, non-physical version of a camera that captures the light within a given field of view incoming to a single point in space instantly. Two reasons that this camera model is non-physical because real cameras have lenses with a finite area and focal length, and the camera also requires a certain amount of exposure time to capture light to form the image.

If we want the images we produce to more closely resemble those that might be taken with a real camera, we can simulate a lens with finite area and focal length, and a non-instantaneous exposure time.

In real cameras, depth of field is caused by a lens's inability to focus light perfectly from all points regardless of their distance to the camera. Simulating a lens with an area and focal length will create a depth a *depth of field* effect: there will be a plane of focus in the image perpendicular to our viewing direction, objects close to this plane will appear sharp and in focus, objects far from the focal plane will be blurred and out of focus. This effect is simulated by generating rays at the origin of the camera, determining where they would intersect the focal plane, then perturbing the origin of the ray by randomly sampling in a small area around the origin (usually in a disc perpendicular to the viewing direction) while ensuring that the ray will still pass through the same point on the focal plane. By tracing many rays from many different points on the simulated lens and averaging out their contributions, we can produce an image with a realistic depth of field effect.

Physical cameras require the aperture to open for a period of time to capture enough light to take a picture. This is called exposure time. When there are fast-moving objects in the scene we are trying to capture, this combined with the finite exposure time, leads to the moving objects having a streaked, blurred appearance, which is commonly referred to as *motion blur*, such as in Figure 2.12. This effect can be simulated by sampling a moment during the exposure time, and rendering the objects with their geometry updated to be in the appropriate place based on their starting position, velocity and the time elapsed. After sampling many points in time and rendering the associated image, we can average these images to produce a final image that has the moving objects appropriately blurred.

Both depth of field and motion blur are effects that effectively require the scene to be rendered multiple times, and the resulting images to be averaged together, which can dramatically increase the amount of time needed to render the desired image. Nevertheless, the resulting effects can be very effective at achieving a particular visual style, and are popular in animated feature films, as seen in Figure 2.13.

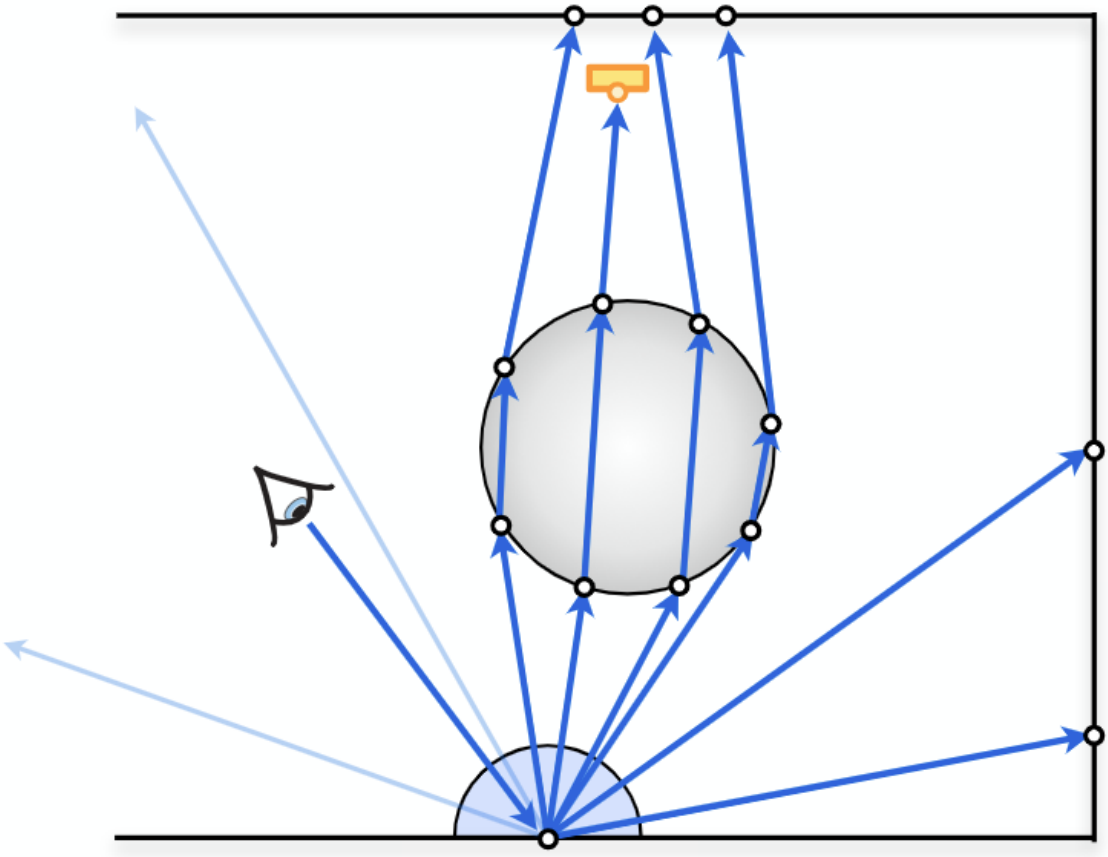


Figure 2.10: The interactions at the surface of the glass sphere in this image are perfectly specular. For an incoming ray, there is a delta BRDF at the surface, which directs the outgoing ray along a unique path. Perfectly specular surfaces like this do not have a diffuse term, which means that we cannot use next event estimation to reduce variance. We can still use implicit path tracing and hope to randomly hit the light source, but this is inefficient if the light source is small. Many rays will miss, leading to a slower rate of convergence. *Source: ECSE 546 Course Notes, Derek Nowrouzezahrai*

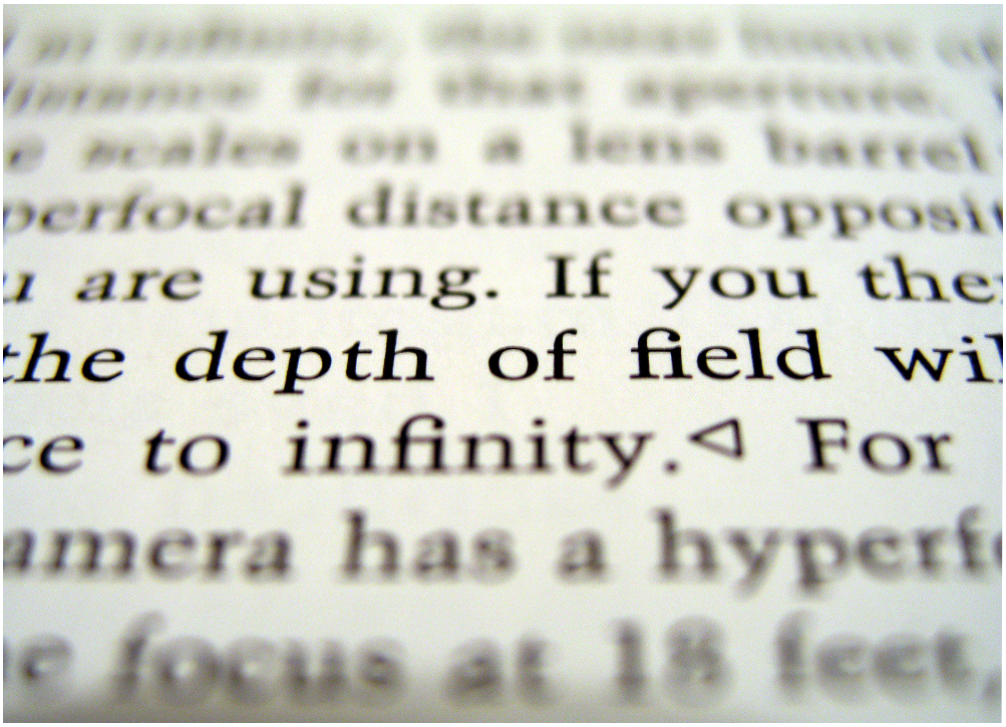


Figure 2.11: An example of depth of field. The focal plane/point in this image lies near the center of the image. The words close to the focal plane are sharp and in focus, while words behind and in front of the plane being out of focus and blurred. *Source: CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=330435*

Figure 2.12: An example of motion blur showing a moving bus and a stationary telephone booth. The high speed of the bus with respect to the shutter speed/exposure time of the camera used results in the bus being blurred into a streak in the direction of motion. *Source: By E01 - originally posted to Flickr as London bus, CC BY-SA 2.0, https://commons.wikimedia.org/w/index.php?curid=4575184*

Figure 2.13: An example of simulated camera effects in Toy Story 4. Depth of field is used to direct the audience's attention to Woody and Bo Peep by having them appear in focus, while the Ferris wheel far in the background is out of focus. The lights on the Ferris wheel also exhibit another effect called Bokeh. *Source: Toy Story 4, https://cdn.fstoppers.com/styles/large-16-9/s3/lead/2019/10/4bf2c77c126d0069cc9d9b9dbda465fc.jpg*

Chapter 3

Differentiable Rendering

Now that we have built up sufficient background on forward rendering, we will examine the challenges that arise when we want to make the rendering process differentiable.

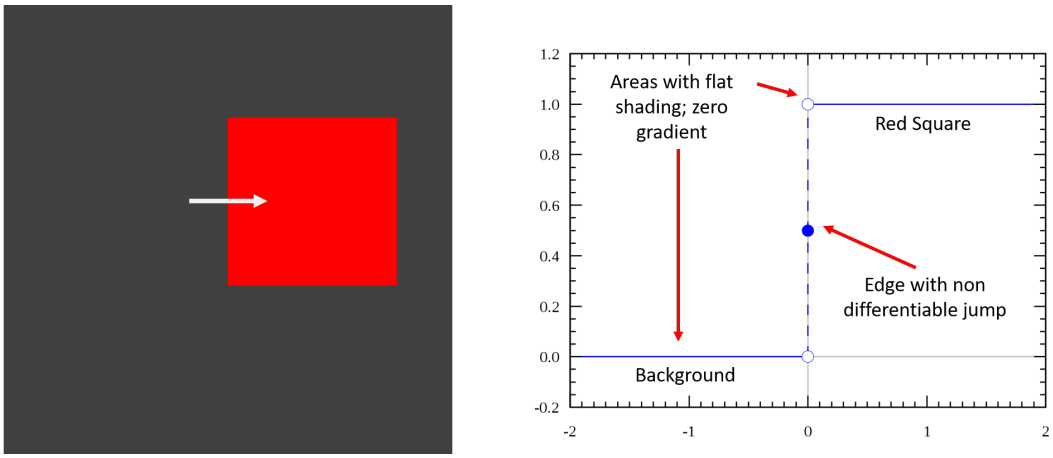
Some parameters of the scene are differentiable without any modifications necessary. Given a differentiable BRDF, the image will be differentiable with respect to the material parameters. Diffuse and Phong are common BRDFs that are naturally differentiable. Similarly, textures are also easily differentiable.

Difficulties arise when we consider the geometry; the placement and the shapes of the objects in the scene. In both rasterization and ray tracing, when considering an image, discontinuities are introduced in two cases: 1. at the edges of objects and 2. where two or more objects overlap and the one closest to the camera must be determined. These two problems are often referred to together as the *differentiable visibility* problem.

3.1 Discontinuities and Differentiability

The first source of discontinues we must consider is the edges of objects. We refer to these as *edge discontinuities*. Consider an image of a red square on a dark background,

as3.1 we cross from the black background onto the red triangle, there is an instantaneous colour change. This jump is a discontinuity that is non-differentiable. Furthermore, on either side of the jump, the shading is completely uniform. There is no indication at the individual pixel level that we are close to an edge. These areas have a gradient of zero, which is not useful if we want to optimize the position of the red square.

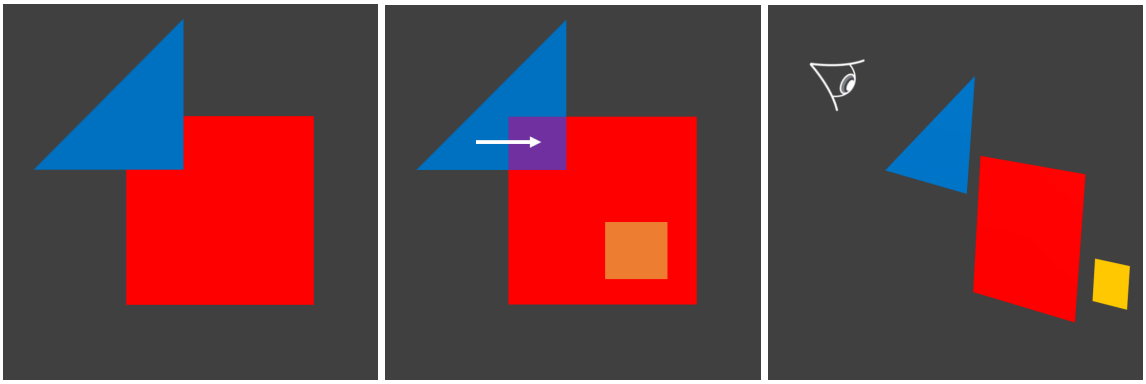


(a) **(b)**

Figure 3.1: The change in colour as we travel along the white arrow in **(a)** is described by **(b)**. The slope of the graph of in **(b)** is the gradient as we travel along the white arrow. Since the shading in both the background and the red square is uniform, the gradient is 0 everywhere, except at the edge where there is a non-differentiable discontinuity.

The second source of discontinuities is when objects in the scene overlap from the viewing position. In normal rendering, the object closer to the camera will occlude the one behind it. The occluded object will not be visible in the resulting image, and all gradients related to the occluded object will be zero. We refer to these as *occlusion discontinuities*. Additionally, we also can have the case where two objects only partially overlap from the viewing position, resulting again in a non-differentiable discontinuity as we cross from viewing one object to viewing the other.

To solve these problems, two modifications have to be introduced. Additional information about the edges of objects and their gradients needs to be computed explicitly or encoded into the rendered image differentiably, and occluded objects in the scene need to



(a) **(b)** **(c)**

Figure 3.2: **(a)** a scene where geometry occludes other geometry. **(b)** The blue triangle partially covers the red square, occluding the area represented by the purple region. As we move along the white arrow, there is a non-differentiable instantaneous jump when we transition from one shape to the other, similar to the one described by 3.1b. Additionally, the orange square represents a yellow square that is completely occluded by the red square. This yellow square would normally have no contribution to the image, and we could not compute any gradients related to it. **(c)** represents the scene as viewed from the side.

have some contribution to the rendered image or have their gradients taken into account otherwise.

3.2 Related Work

In this section, we discuss some of the most influential and recent works in differentiable rendering. Several different methods for differentiable rendering have appeared in recent years, for both rasterization and ray tracing. Some methods handle both the edge and occlusion discontinuities described above, while others focus only on edge discontinuities.

3.2.1 Rasterizers

For rasterization-based methods, several different formulations have appeared in the literature recently [5, 6]. Rasterization differentiable rendering methods typically have a focus on applying their systems to interesting ML applications such as mesh reconstruction, lighting model estimation, or texture reconstruction.

OpenDR

One of the earliest differentiable rasterizers published was OpenDR [4]. It computed approximations to the gradients using Taylor expansions. However, these gradients are non-zero in only a small region around the edges of the geometry.

Soft Rasterizer

In Soft Rasterizer [5] (SoftRas), each triangle is viewed as a probability distribution, which decays smoothly from the center of the triangle after it has been projected onto the image. This has the effect of making all triangles have some contribution to each pixel on the image place, with the amount of contribution to distant triangles being modulated by the rate of fall off of the distribution, which can be controlled using a parameter. The depth ordering of the triangles for a given pixel is a Softmax-like weighted sum based on the projection distance from the geometry to the image plane (z-buffer).

Differentiable Interpolation Based Renderer (DIB-R)

DIB-R [6] focuses on interpolating vertex information from vertices of the closest triangle to the image plane for each pixel. This vertex-based approach has some nice advantages as one of the core elements for rasterization is the vertex, which can carry information regarding the position, colour and even texture of the surface it resides on.

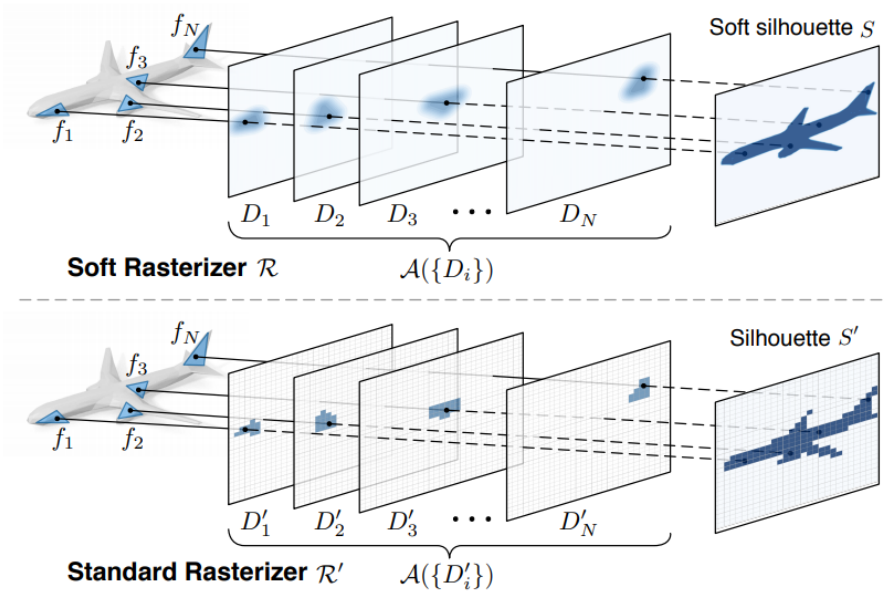


Figure 3.3: The Soft Rasterizer [5] $\mathcal{R}$ (top) can render a mesh silhouette that closely resembles the one generated by a standard rasterizer $\mathcal{R}'$ (bottom). The standard rasterizer renders a pixel as solid if it is covered by a projected triangle. This is a discrete and non-differentiable step. By approximating the rasterized triangles $\{\mathcal{D}'_i\}$ with a "soft" continuous representation $\{\mathcal{D}_i\}$ based on signed distance field and combining $\{\mathcal{D}_i\}$ using a differentiable aggregate function $\mathcal{A}(\cdot)$ the process becomes differentiable. *Source: Liu et al. [5]*

However, differentiable renderers based on rasterization have several limitations. They are only able to handle effects related to primary visibility. They do not take into account effects such as shadows, reflections, and indirect light. Additionally, most of the recent rasterization-based differentiable renderers are only able to handle single objects.

To handle more complex light transport effects and multiple objects, there exists a different class of differentiable renders based on ray tracing techniques.

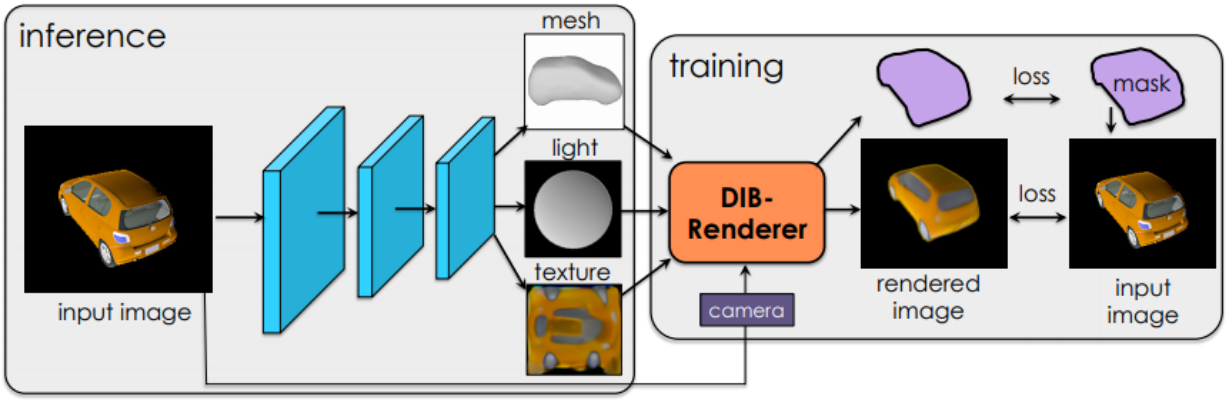


Figure 3.4: Chen et al. [6] use their differentiable renderer to train an ML system to predict geometry, texture and lighting given an input image. During training, the predicted parameters are rendered with a known camera. The 2D image loss between the input image and rendered prediction is used to train the prediction network. prediction networks. *Source: Chen et al. [6]*

3.2.2 Ray Tracers

These types of differentiable renderers can be more broadly classified under the umbrella of differentiable physics, as they are concerned with differentiating the physically-based transport of light in a scene. A few differentiable Monte Carlo ray tracing techniques have been presented.

Differentiable ray tracing allows for global illumination (i.e. inclusion of indirect lighting) and other effects such as refraction, specular reflections and shadows. Detailed, photorealistic images with these effects can be produced, we are still able to backpropagate gradients through these effects, allowing for applications based on these effects to be explored.

Redner

In 2018, Li et al. [8] published a general-purpose differentiable ray tracing method with no approximations in the gradients. Li et al. released code alongside their publication, in a system they called REDNER (render backwards). This system introduces a technique they call edge sampling to compute gradients of the integration terms. Essentially, they construct the problem so that rather than having to directly compute the gradients of the Monte Carlo integrals, which would be quite challenging, they use the fact that integrals and gradients commute under suitable conditions. Recognizing this property, they compute the integral of the gradient, which can be achieved through another MC integration and by careful construction of the integrand.

This method requires additional sampling to determine the location of the edges for both primary visibility and secondary visibility. For primary visibility, the edges can be found relatively easily by recomputing the silhouettes of objects from the camera's viewing position. For secondary visibility (i.e. light paths that contain bounces), it requires the use of complex data structures to keep track of the edges visible from the perspective of the shading point.

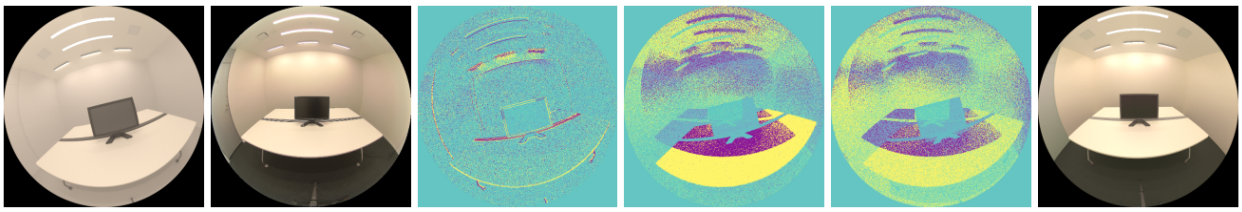


Figure 3.5: The edge sampling method computes gradients from the output image for various scene parameters, such as camera pose **(c)**, material parameters **(d)**, and lighting parameters **(e)**. The gradients are used in an inverse rendering task to fit an initial approximation of a scene **(a)** to match a real photograph **(b)**, with the result appearing in **(f)**. *Source: Li et al. [8]*

Path Space Differentiable Rendering

This method [12] established a formulation of differential rendering that allows for bidirectional path tracing. Previous methods were only capable of handling unidirectional path tracing. In theory, using bidirectional methods would allow for more complex light transport scenarios. The gradients computed using this method are unbiased and avoid the costly silhouette samplings that Redner uses.

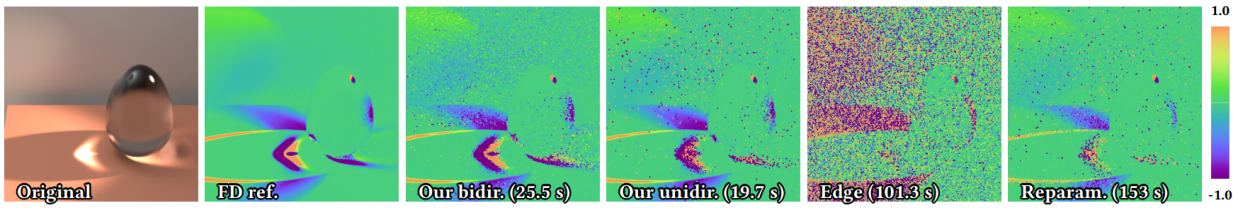


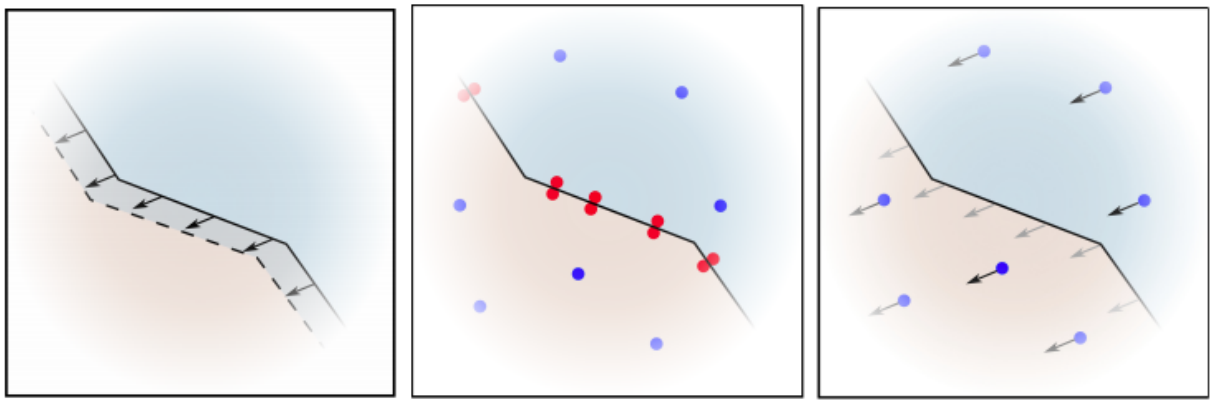
Figure 3.6: The Veach Egg relies heavily on bidirectional path tracing to produce a convincing image efficiently. The gradients for the original image computed with Finite Differences (FD), the bidirectional and unidirectional methods from Zhang et al. [12], Redner [8], and Mitsuba 2 [10] are shown. *Source: Zhang et al. [12]*

Mitsuba 2

Mitsuba 2 [9] [10] uses a reparametrization of the Monte Carlo integral for the rendering equation. This reparametrization causes sampling discontinuities to move with the edges of geometry, which makes shifts in geometry result in continuous changes in the image and admits the computation of meaningful gradients. However, the gradients computed using this method are biased.

This method used here does not focus on finding and sampling the boundaries of objects explicitly. Instead, it casts additional rays to sample the area inside the silhouette of an object and uses heuristics to determine where the edges of the silhouette lie. While these heuristics work well in general, they are some situations where they fail. This leads to approximations and inaccuracies in the gradients calculated, which causes bias. How-

ever, this method is faster than those that rely on explicit edge finding, and the gradients are sufficiently accurate to be useful in typical situations.



(a) Integrand with discontinuity

(b) Edge sampling from Li et al. [8]

(c) Change of variables from Loubet et al. [10]

Figure 3.7: (a) Integrands in physically-based rendering have discontinuities whose location depends on scene parameters such as object positions. **(b)** In Redner [8], gradients are estimated by sampling pairs of paths along the silhouette edges of geometry (red dots) in addition to standard Monte Carlo samples (blue dots) for determining appearance. **(c)** In Mitsuba 2, the expensive sampling of visible silhouette edges from Redner is avoided by using a change of variables that ensures the discontinuities are fixed with respect to the Monte Carlo samples for infinitesimal changes of scene parameters. These MC estimates can be differentiated using automatic differentiation. *Source: Loubet et al. [10]*

Radiative Backpropagtion

Another ray tracing based technique called Radiative Backpropagtion has also been recently published [11]. This technique offers a significant speed improvement compared to the reparameterization technique. This technique considers the entire scene as a single transport function and computes an estimate for this function. They then use an adjoint method to compute gradients. This approach allows them to avoid having to keep a computation graph of all operations done to construct the image, saving a substantial amount

of memory and time as they don't have to do a backward pass through the graph. However, currently, this technique has some significant limitations. Primarily, it does nothing about the differentiable visibility problem, and therefore it does not allow for changes of objects' positions or vertices.

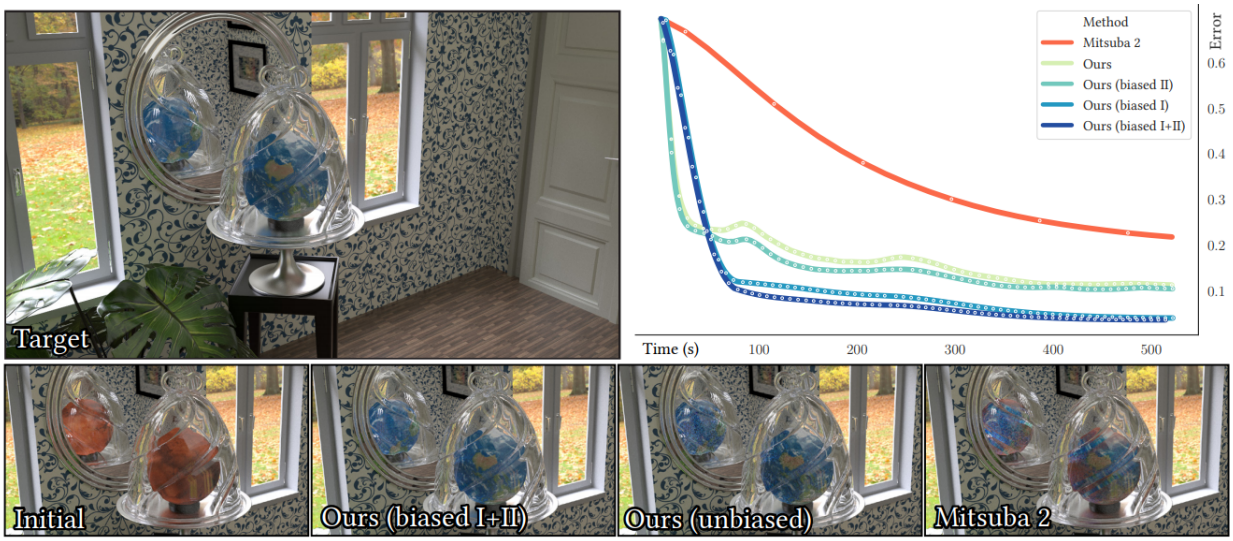


Figure 3.8: The radiative backpropagation method reconstructs the texture of a globe seen through a glass jar. The globe is initialized with a Mars texture, and the system attempts to match the target image by differentiating scene parameters using the L2 distance to the target. The plot shows the convergence over time comparison with Mitsuba 2 and the different variants of radiative backpropagation introduced. The radiative backpropagation method demonstrates speedups of up to $\sim 1000\times$ over Mitsuba 2. *Source: Nimier-David et al. [11]*

Unbiased Warped-Area Sampling for Differentiable Rendering

This method proposed by Bangaru et al. [21] uses the divergence theorem to formulate an expression for the derivative at the boundary of a shape based on samples taken over the area interior of an object. Deriving the area integral from first principles, they can formulate consistent and unbiased estimators for the gradients.

This method still requires the tracing of auxiliary rays. This method is similar to the method of Loubet et al. [10], which also relies on samples based on the interior of the object but improves upon it by producing unbiased gradients. They also show that the Loubet et al. method is a special case of their method.

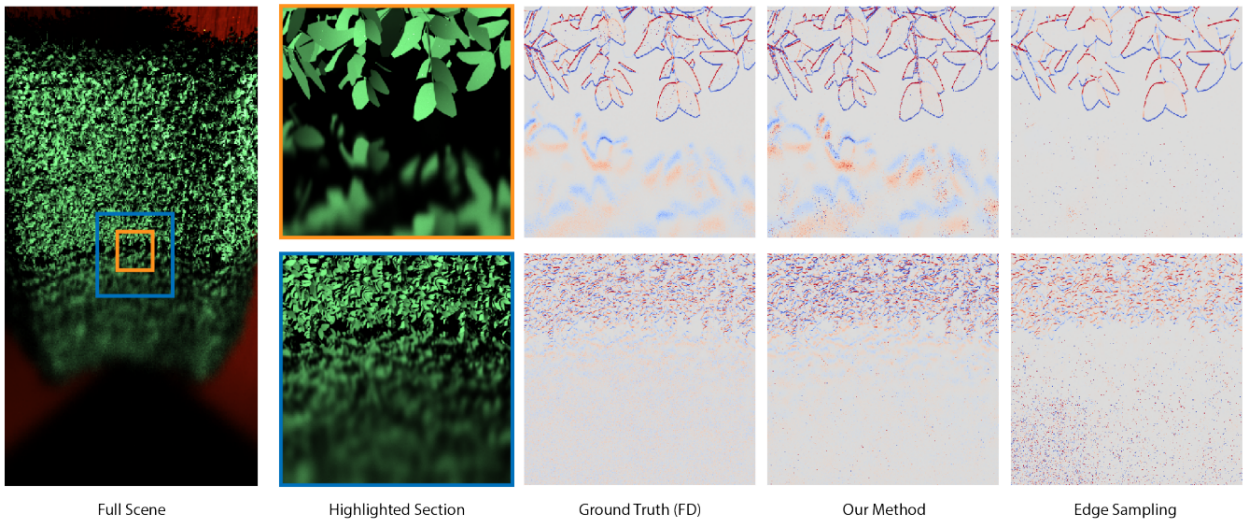


Figure 3.9: In this scene, a leafy wall is reflected in a glossy black surface. Bangaru et al. [21] develop an unbiased estimator that computes the boundary contribution to gradients from area samples, to produce accurate, low variance gradients. Compared to the edge sampling method [Li et al. 2018] which needs to explicitly sample boundary points (difficult in the glossy reflection area), the warped area sampling method can use samples from a standard path tracer. We can see in this scene the Bangaru et al. [21] (labeled 'Our Method' above) computes gradients much more than the edge sampling method. *Source: Bangaru et al. [21]*

3.3 Automatic Differentiation

One technology, in particular, has helped enable the recent progress in differentiable rendering: automatic differentiation. Automatic differentiation, also called auto diff or AD, is a feature that allows a user to numerically evaluate the derivative of a function specified by a program.

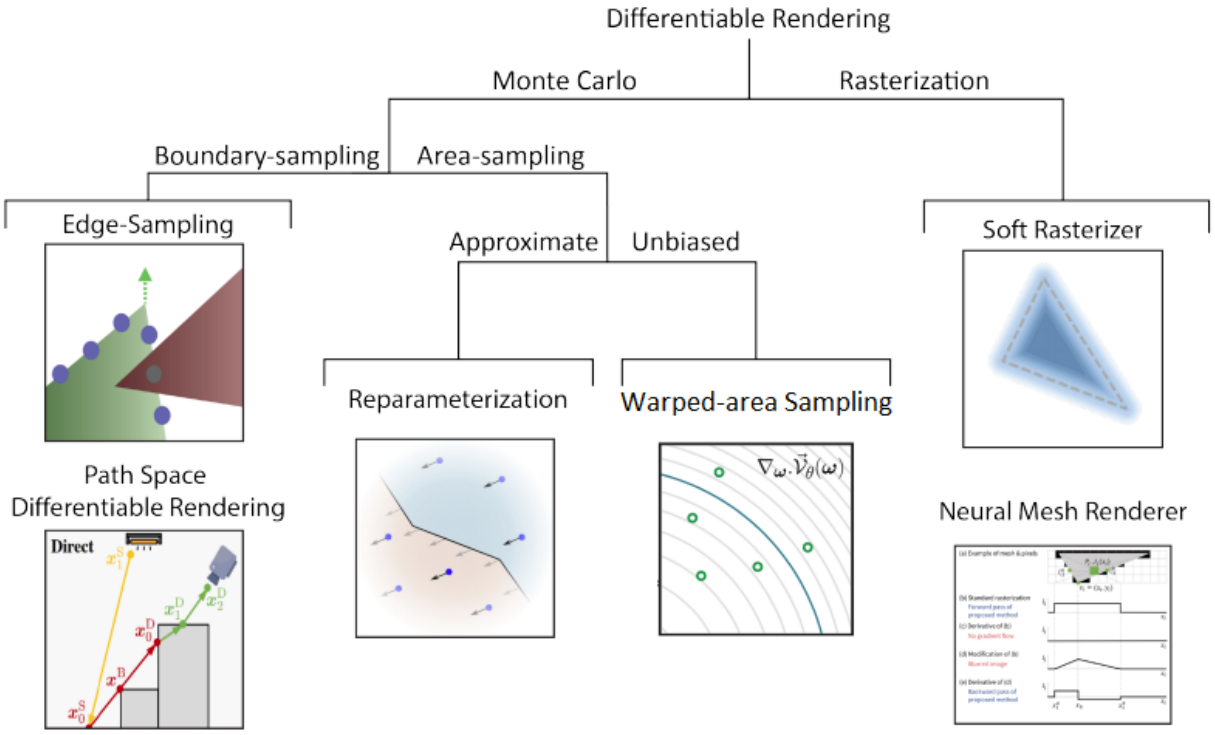


Figure 3.10: A family tree of differentiable renderers. Both of the boundary sampling techniques require an importance sampling data structure. The Redner technique from Li et al. [8] uses a 6D Hough tree to track silhouettes and Zhang et al. [12] pre-compute a spatio-angular photon map. The reparameterization method from Loubet et al. [10] does not need a data structure. It computes rotations as needed during the standard Monte Carlo rendering process. However, this method is biased. The Warped Area Sampling from Bangaru et al. [21] has the simplicity and flexibility similar to the reparameterization method while being unbiased. *Source: Bangaru et al. [21]*

After writing a function using a library that supports automatic differentiation, the program can automatically calculate the gradient of this function with respect to specified parameters. Typically, the library keeps track of all the operations done on the input to compute the output, storing the derivatives at each step in a tree or graph structure. These individual derivatives can then be used in combination with the chain to calculate the full gradients between the input and outputs.

Auto diff is a practical and efficient alternative to calculating by hand analytic gradients for whatever arbitrary function a user might want to implement. In the context of differentiable rendering, rendering a scene can be thought of as a function. This function is complex and it would be impractically difficult to calculate gradients by hand for the majority of the interesting inputs (such as geometry), making automatic differentiation a critical component of a differentiable renderer.‘

Some examples of popular libraries that offer automatic differentiation are PyTorch [22, 23] and Enoki [24].

Automatic differentiation has found its principal use in machine learning applications, where gradients are used in optimization algorithms such as Adam [25] for updating the weights of the model to improve its accuracy.

This technology also sparked interest in other has applications in other areas, such as differentiable physics. Differentiable rendering can be considered a special case of differentiable physics, that is focused on physically-based light transport.

3.4 Applications

A direct application of differentiable rendering is in inverse rendering. In general, inverse rendering is inferring detailed scene parameters from an image, similar to computer vision, the main difference being is that there is a rendering system somewhere in the loop. For example, given an approximate scene description and a target image, the difference between the target image and the rendered image of the approximation can be computed. From this difference, gradients with respect to the shape or position of objects can be computed, and we can use gradient descent methods to minimize the difference, and update the approximation to more closely match the target image.

Differentiable rendering has also been integrated into various machine learning pipelines. Differentiable rendering is attractive in ML contexts because of its ability to pass gradi-

ents through the rendering process. The computation of gradients is central to training models.

One example unsupervised mesh reconstruction. In these applications, the goal is to train a model to output a mesh (or another 3D representation) from one or several photos of an object, such as a chair, without any references for what the 3D output should be. The training loop in these applications leverages a differentiable renderer to render the image of the output mesh, and compare this image to the original input image. Ideally, we want them to match, however, initially, the model will be bad at predicting the 3D representation. We can calculate loss between the output image and the target/input image, and back-propagate the gradients from this loss through the differentiable renderer to update the parameters of the model.

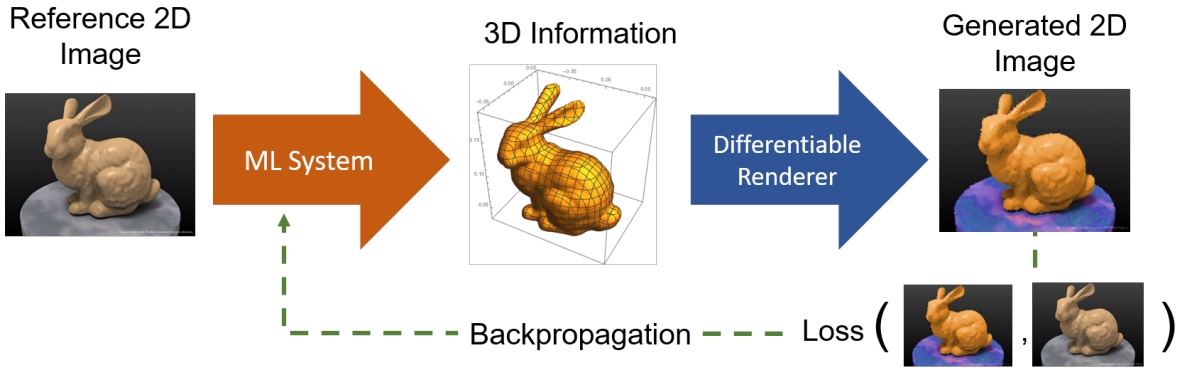


Figure 3.11: An example of how a differentiable renderer can be used in the training loop for an ML model. We are training the ML system to extract 3D information (ex. a mesh, object orientation, light position) from an image. By including a differentiable renderer in the loop, we can render an image based on the 3D information we extract and compare it to the reference image, calculate a loss, and update the parameters of the model. The advantage of this unsupervised method is that it does not require the reference 3D information associated with the image. If we were to use a non-differentiable renderer, it would not be possible to backpropagate meaningful gradients through the rendering process.

Another example is adversarial image generation for testing the robustness of computer vision or classification systems. Here, we can start with an image of a chair, which will be correctly classified by the model. By calculating gradients, we can determine how to modify the image we send to the model, to trick it into misclassifying the chair as another object. Often, small, nearly imperceptible changes can cause difficulty for these classification models, causing them to classify objects that humans can easily recognize, as something absurd [26, 27], such as in Figure 3.12.

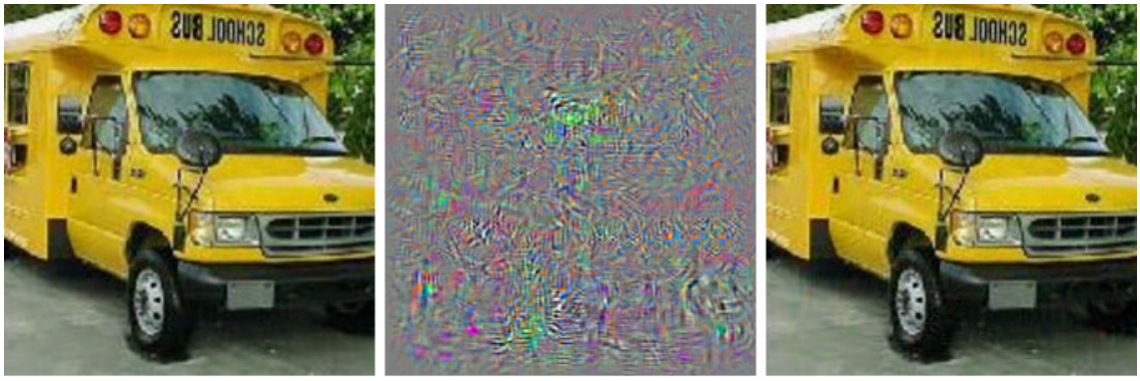


Figure 3.12: The left image of the school bus is correctly classified by AlexNet, a computer vision model based on deep neural networks. Adding the middle image to the left image causes the right image to be classified as an ostrich. This example is taken from [26]

Chapter 4

Methodology

In this section, we discuss the novel algorithm for differentiable rendering that forms the body of this thesis.

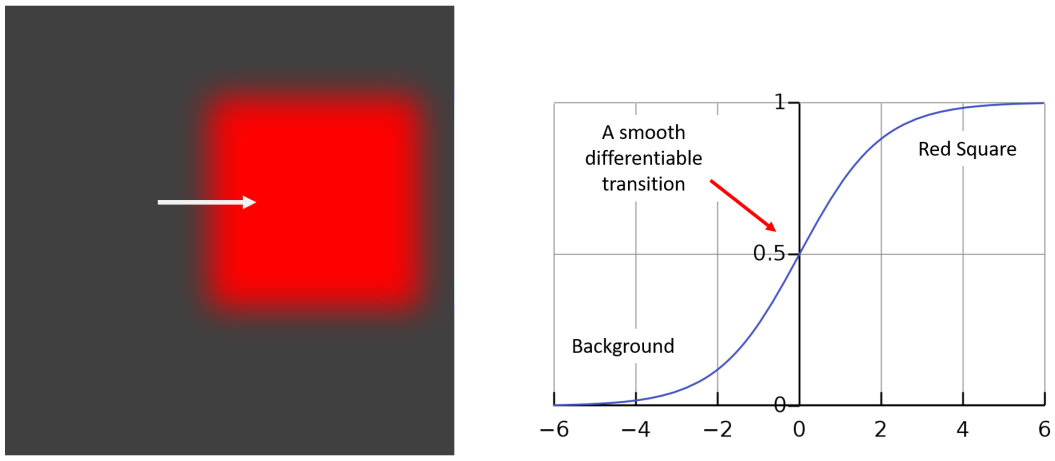
This work was mainly inspired by the approach taken in SoftRas [5], but we extend their idea of considering triangles as a probability distribution, to work in a path tracer. This allows us to handle multiple bounces and other more complex light transport effects that are typically unachievable with rasterization.

We also explore how differentiable rendering could be used in inverse rendering problems involving some camera effects, namely motion blur and depth of field.

An advantage to our approach compared to other global illumination differentiable renderers (such as Redner [8] and Mitsuba 2 [9]) is that it does not require casting additional rays or taking additional samples to calculate gradients.

4.1 Edge Discontinuity

In our system, we view triangles as a probability distribution in 3D space, rather than just in the image plane as proposed in SoftRas. With this approach, a ray can intersect with a triangle in the entire plane of the triangle. By considering the triangles as distributions, the edges of the triangles are blurred or smoothed out from being sharp, non-differentiable discontinues, to being smooth continuous functions. A simple example of how this appears visually is given in Figure 4.1. This blurring effectively regularizes the discrete step that would be present at the edges of geometry in the scene and admits the computation of useful gradients. This smoothed function can be handled by automatic differentiation techniques.



(a) **(b)**

Figure 4.1: The change in colour as we travel along the white arrow in 4.1a is described by 4.1b

This blurring function, which we will denote as α, effectively modulates the transparency of the triangle.

Given a point in 3D space x, the value of α is given by Equation 4.1:

$$\alpha(x, \sigma) = \text{sigmoid}\left(\text{sign}(d(x))\frac{d(x)^2}{\sigma}\right), \tag{4.1}$$

where the point x lies in the plane of triangle we are computing the transparency for. $d(x)$ is the signed distance (positive inside the triangle) to the nearest edge of the triangle (in the plane of the triangle), and σ is a value used to control the edges' sharpness. The sigmoid(z) function is given by Equation 4.2.

$$\text{sigmoid}(z) = \frac{1}{1 + \exp(-z)} \tag{4.2}$$

Decreasing the value of σ decreases the amount of blur in the image, making the image edges in the image sharper. In the limit, where $\sigma \to 0$, we recover the discrete step we have in traditional rendering. In practice, setting σ to a small value (on the order of $1\text{e}{-7}$) results in an image that does not have any perceivable blur.

This results in the triangle having the highest opacity in its interior, having $\alpha(x) = .5$ at an edge, and decaying asymptotically to 0 as the distance to the edges goes to infinity. $\alpha(x)$ is a smooth and differentiable function of the vertex positions of the triangle that directly contribute to the value of the pixels, which is necessary for the computation of gradients with respect to the vertex positions.

As a result of this formulation, a ray will produce a valid intersection with a triangle anywhere it intersects the plane the triangle lies in. If we intersect the plane far outside the boundaries of the triangle, the opacity will be very low, but still a valid intersection nonetheless. Traditional renders take the closest valid intersection along a ray. This does not work for our proposed system, as this could result in just a single triangle whose plane lies close to the origin of the ray appearing in the image, even if the intersection with the plane is so far from the triangle that the α value at this point will be effectively 0, and the triangle will not be visible.

We will need to devise a way to determine how to display the contributions from all the triangles along the ray. This leads us directly into how we handle occlusion discontinuities.

4.2 Occlusion Discontinuity

To handle the discrete operation that results from determining the surface nearest to the camera, we again use a similar approach as taken in SOFTRAS [5], which uses an aggregation function based on the Softmax operation. We determine the colour of pixel I_j using a weighted sum to blur together the contributions from all triangle k along the path of the ray (Equation 4.3).

$$I_j = \sum_k w_k(z_k, \alpha, \gamma)\, S(x_k) \tag{4.3}$$

with

$$w_k(z_k, \alpha, \gamma) = \frac{\alpha(x_k, \sigma) \exp\left(z_k/\gamma\right)}{\sum_j \alpha(x_j, \sigma) \exp\left(z_j/\gamma\right) + \exp(\epsilon/\gamma)} \tag{4.4}$$

The weight w_k of triangle k is based on the distance to the camera z_k and the opacity of the triangle $\alpha(x_k, \sigma)$ at the point of intersection. γ controls the weighting of the surfaces. Decreasing γ decreases the amount of blurring over the depth, as small γ weights the closer surfaces more heavily. ϵ is a small constant for numerical stability. This weighted sum removes the discrete operation of determining the closest surface and replaces it with a continuous and differentiable sum operation. By construction, all the weights along the ray will sum to 1. This ensures that no additional colour is being added to the pixel. The function is also monotonically decreasing as z_k increases, which ensures that the closer surfaces are weighted more heavily than surfaces that are further away.

Introducing this sum over all the triangles intersected along a ray creates another difficulty. Computing the shading $S(x_k)$ for every triangle k intersected by a given ray requires casting additional rays to determine the direct and indirect lighting. Doing this for multiple surfaces, for multiple bounces, would result in an exponential increase in the number of rays being cast. This is undesirable as the computation power needed

would exponentially increase with the number of triangles in the scene and the number of bounces considered.

We propose a numerical method to approximate this sum while using a constant number of rays. We use importance sampling [28] to estimate Equation 4.3. We importance sample one surface along the ray based on the weight of its contribution w_k in the sum, and trace additional rays for determining shading for *only* this surface. Since we pick only a single surface to compute shading for, we can keep a constant number of rays.

This sampling step introduces some additional noise in areas where objects are overlapping. This noise is reduced by having multiple samples per pixel, which is already something that is required in regular ray tracing. The importance sample is based on the weighting of the surface and the value of α at the shading point, which helps to reduce the variance of the estimate.

$$\sum_k w_k(z_k, \alpha, \gamma)\, S(x_k) \approx \frac{w_i(z_i, \alpha, \gamma)\, S(x_i)}{p(i)} \tag{4.5}$$

with

$$w_k\, \alpha_k \sim p(i) \tag{4.6}$$

4.3 Modifications to the Path Tracing Algorithm

In this section we breakdown more precisely how the blurring parameter α and the depth weighting function w_k are used to modify the standard path tracing algorithm.

Given the point of intersection x on the triangle in the scene seen when casting a ray through pixel I_j, the colour of the pixel should be given by the following Monte Carlo estimator for the rendering equation [14]

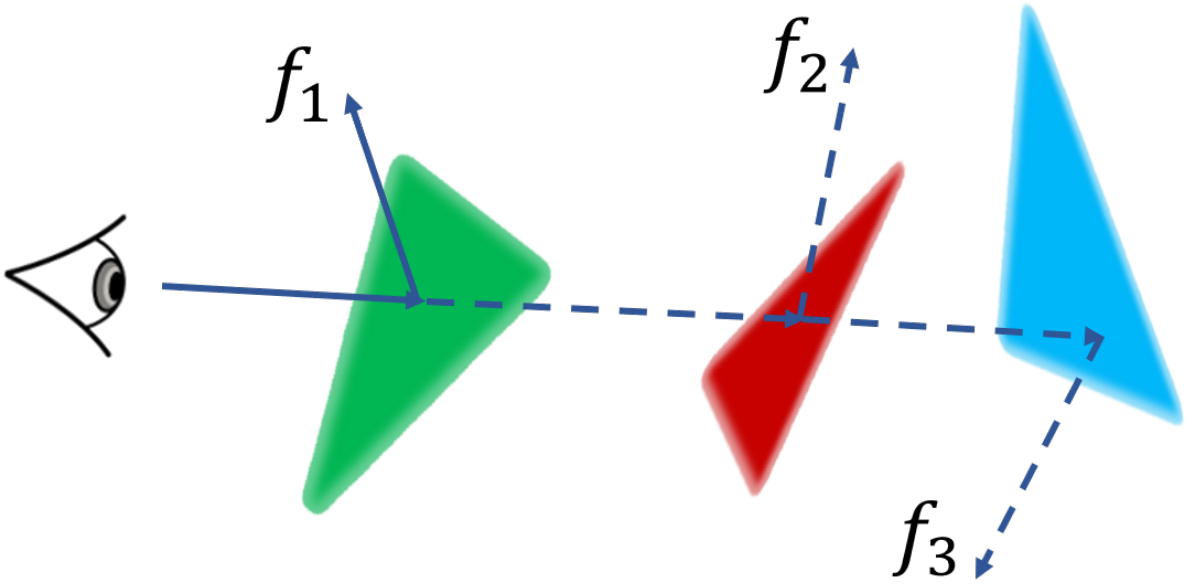


Figure 4.2: Since all the surfaces along the ray are transparent, they all contribute to calculating the colour seen by the eye. Calculating the shading f_i at each triangle would involving casting additional rays for each surface, leading to an exponential increase in the number of rays being traced. We instead sample one surface and weight its contribution to produce a one sample estimate for the sum.

$$I_j = L_e(x, w_o) + \frac{1}{N}\sum_{k=1}^{N} \frac{f(x, w_i^k, w_o) L_i(x, w_i^k, w_o)(n \cdot w_i^k)}{p(w_i^k)} \tag{4.7}$$

This is the Monte Carlo estimator for the rendering equation. The indirect illumination integral is estimated using a Monte Carlo integration. When path tracing, the Monte Carlo integration term uses one sample. Which simplifies the equation to:

$$I_j = L_e(x, \omega_i, \omega_o) + \frac{f(x, \omega_i) L_i(x, \omega_i)(n \cdot w_i)}{p(w_i)} \tag{4.8}$$

An image rendered with only a single path traced for each pixel is very noisy. To reduce the noise in the image we cast multiple rays per pixel and average their contributions. This effectively amounts to finding many light paths originating from the pixel.

$$I_j = \frac{1}{N} \sum_{k=1}^{N} \left[L_e + \frac{f(x, \omega_i^k, \omega_o^k) L_i(x, \omega_i^k)(n \cdot \omega_i^k)}{p(\omega_i^k)} \right] \tag{4.9}$$

Equation 4.9 for I_j contains implied discrete operations that determine the edges of geometry and which objects appear in front of others. We need to do some reformulation to remove these implied discrete operations included here.

In Equation 4.10, we modify the single sample MC estimator from Equation 4.8. We replace the discrete step of determining the closest surface (i.e. the visible surface that is shaded) with a sum over all triangles along a ray. This ensures that triangles behind the closest one will have some contribution to the radiance travelling along the ray and the output image. Determining the edges of triangles (essentially, checking if a ray intersects the interior of a triangle) is replaced by the $\alpha(x, \sigma)$ term, which smoothly modulates the intensity of the triangle over the plane the triangle lies in. These changes effectively remove the discrete components from the computation of the image. These can be applied recursively for bounces at greater depth.

$$I_j = \sum_{k=1}^{S} w_k \alpha_k \left[L_e^k + \frac{f_k(x_k, \omega_i^k, \omega_o^k) L_i(x_k, \omega_i^k)(n_k \cdot \omega_i^k)}{p(\omega_i^k)} \right] \tag{4.10}$$

where the sum is over S, which is the set of triangles intersected along the ray, α_k is explained in Section 4.1, and w_k is the weighting function shown in Equation 4.4.

To reduce the noise from the single sample in Equation 4.10, we can cast multiple samples per pixel to reduce the noise, as in Equation 4.9.

To avoid calculating every term in the sum for Equation 4.10, we use a one-sample Monte Carlo to calculate an estimate for it. This Monte Carlo sampling selects a single surface s which we calculate the shading for by tracing subsequent rays from it.

$$I_j \approx \frac{w_s \alpha_s}{q(w_s, \alpha_s)} \left[\frac{L_e^s + f_s L_i^s (n_s \cdot \omega_i^s)}{p(\omega_i^s)} \right] \tag{4.11}$$

The surface k is randomly sampled from the multinomial distribution q which is constructed by normalizing the product of the weight and transparencie,

$$q(w_i, \alpha_i) = \frac{w_i \cdot \alpha_i}{\sum_{k=1}^{S} w_k \cdot \alpha_k} \tag{4.12}$$

4.4 Explicit Connections and Shadows

To reduce noise in the rendered image, we implement explicit path tracing. For each shading point, an emitter is selected with a uniform probability. A point on the emitter is selected uniformly over its interior (i.e. no blur applied) area. A shadow ray is then traced to determine visibility between the shaded point and the selected point on the emitter.

Visibility is computed as *transmission* $\prod_{i=k}^{S}(1 - \alpha(x_k, \sigma))$ along the shadow ray for intersections k that occur between the shaded point and the emitter. This factor attenuates the emitted radiance L_e. The attenuated radiance is then modulated by the Bidirectional Reflectance Distribution Function (BRDF) and the cosine foreshortening term to calculate the shading at the point. The contribution is then weighted by the probability of selecting the point on the emitter.

Since visibility in these shadow rays is differentiable due to our modifications remove the edge discontinuities, this allows computing gradient signals from the shadow of an object.

4.5 Material Model

We use a simple mixture BRDF model that has both diffuse (Lambertian) and specular (Phong) components [17]. The advantage of this material model is that it is simple to implement, can produce a range of materials with a good appearance and has no discontinuities in it. This makes it naturally differentiable and easily handled by automatic differentiation. The BRDF is given in Equation 4.13:

$$f_r(x, \omega_i, \omega_o) = \rho_d \frac{1}{\pi} + \rho_s \frac{n+2}{2\pi} \max(0, \cos^n(\alpha)) \tag{4.13}$$

where ρ_d is the diffuse albedo (reflectivity), ρ_s is the specular albedo, n is the Phong exponent (higher values yield a more mirror-like specular reflection) and α is the angle between the direction of perfect specular reflection of ω_o and the direction of the incoming lighting.

4.6 Textures

We implemented basic texture mapping for meshes. Textures are can be loaded from an image file, which gets converted to a tensor of size $(H \times W \times 3)$ for the associated mesh. When an intersection point is found on the mesh, we sample the texture using the UV coordinates of the triangle's vertices to obtain an albedo value. We use bilinear interpolation when sampling the texture to improve the appearance of the mapped texture.

The values stored in the texture map and the bilinear interpolation step are naturally differentiable with no special attention needed. To handle areas outside of the triangle, the texture is wrapped to cover the extended area.

4.7 Camera Effects

Camera effects can be added to rendered images to simulate features that arise from the physical properties of a camera and lens. We implement two of the most common camera effects differentiably, namely depth of field and motion blur. To our knowledge, ours is the first work to include differentiable camera effects.

4.7.1 Depth of Field

Depth of field is an effect that results from the focal length of the lens and the size of the aperture in physical cameras. It results in objects far away from the focal plane being blurred. This effect is simulated by displacing (often called jittering) the origin of the rays traced from the camera in the plane perpendicular to the view direction while ensuring that the rays intersect the focal plane at the same point as if the origin had not been altered. Several of these jittered images are rendered and averaged together to achieve the effect. This averaging operation is naturally differentiable.

4.7.2 Motion Blur

Motion blur is an artifact that occurs in images when objects are moving due to the finite shutter speed of the camera. It results in a drawn-out blur of the object along the direction of motion. Motion blur is achieved by assigning objects a velocity, sampling the scene at multiple time steps and averaging the resulting rendered images. This averaging operation is naturally differentiable. We limit our implementation to only linear velocities.

4.8 Caustics and Light Tracing

In our renderer, we implement light tracing and simplified refractive materials. The refractive materials we implement are limited to being only a surface (i.e. they have no thickness or volume), where a purely refractive event takes place (i.e. no light is scattered back in the incident direction). The materials are also completely clear, having no tint or colouration to them. They are only described by their index of refraction (IOR), and a normal map if desired. To relate the value of the pixels in the caustic to the material that casts them, we multiply the radiance passing through the glass pane by the α value. This ensures that the shading of the caustic is directly parametrized by the vertices of the glass material, and we can compute gradients for the position of the glass material based on the caustic.

This simplified material model is sufficient to simulate caustics, an optical phenomenon that results from the focusing of energy by a refractive material such as glass. Visually, this manifests as an area of brightness. We can combine this simple material with a normal map on the surface to create interesting caustics.

One drawback of using light tracing with a point camera model is that purely specular interactions (such as refraction) are unable to connect to the camera. That is to say, they have a probability 0 of randomly interesting the camera. This results in glass and mirrors appearing completely black when viewed directly by the camera.

To overcome this, we render an image using forward path tracing (i.e. starting from the camera), handling the refractive events appropriately, then we combine the light traced image and forward path traced image. The two images are combined using a simple averaging weighted by the number of samples done for each pixel. For pixels that received no samples in the light tracing pass (appearing completely black), the light traced image has weight 0 and the path tracing image contributes with a weight of 1. For pixels where the caustics appear, there are a significant number of light traced samples landing there, which ensures that the light trace image is weighted significantly there.

One drawback of this weighting scheme occurs when there is a significant mismatch between the number of samples done with each pass. If a very large number of SPP are done for the forward pass image with relatively few light paths traced, the forward image contribution will dominate the light traced image, resulting in the caustics appearing significantly dimmer or not at all.

An additional simplifying assumption is made during the forward pass. When computing the shadow rays for the explicit connections to the light sources, refractive surfaces are assumed to be completely transparent and not refract light passing through them, which means they are effectively ignored to compute shadow ray visibility. This is sometimes called a ”straight line” approximation.

Chapter 5

Results

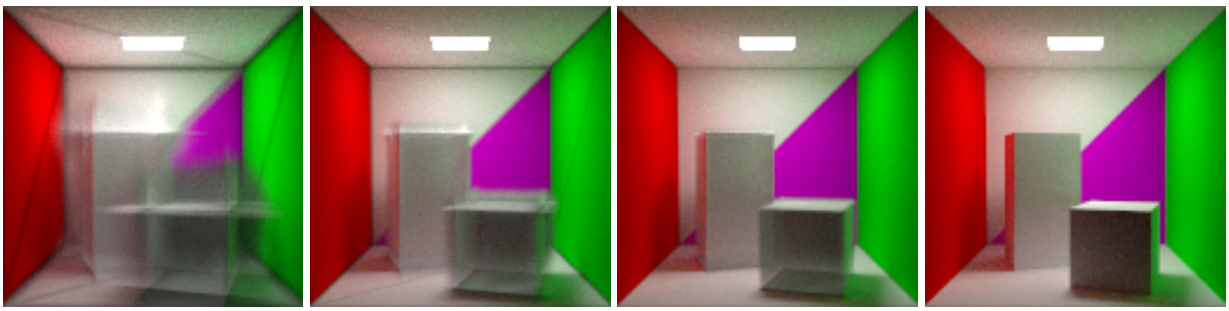
We implemented our differentiable renderer in Python making use of the PyTorch library [23]. Using PyTorch conferred several advantages: parallelization on the GPU by implementing ray tracing as tensor operations, built-in automatic differentiation, optimization tools, and easy integration into Python-based machine learning pipelines. Other people also seemed to have realized these advantages, and during the development of our project, several rasterization focused PyTorch based differentiable renderers appeared, including PyTorch3D [29] and Kaolin [30]. These packages focus more on offering more complete tool sets for loading and manipulating 3D data, and use previously discussed methods for their differential rendering step. PyTorch3D implements a method based on SoftRas [5], and Kaolin implements DIB-R [6].

5.1 Forward Render Examples

In Figures 5.1 – 5.2, we present example images that show the effects of modifying the edge blurring parameter σ and the depth blurring parameter γ. The images are rendered with 5 bounce global illumination with Russian roulette termination and 1024 SPP.

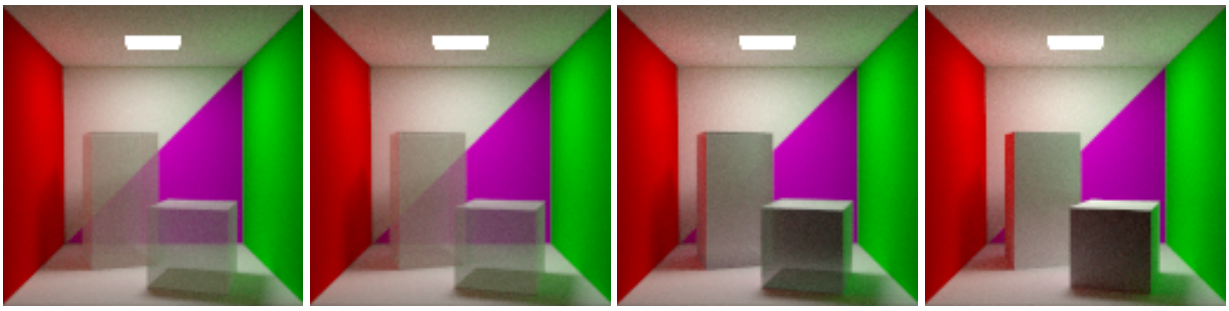
As γ decreases, the closer surfaces are weighted more heavily. One of the triangles that composes the back wall is set to be magenta, to provide some contrast when visualizing the effect of different values of γ.

As σ decreases, the blurring of the triangles decreases. When the values are sufficiently small, the rendered image has few observable artifacts when compared to an image generated with standard rendering techniques.



(a) $\sigma = 1e{-}3$ **(b)** $\sigma = 1e{-}4$ **(c)** $\sigma = 1e{-}5$ **(d)** No blur

Figure 5.1: The blur around the edges of the triangles decreases as σ decreases.



(a) $\gamma = 1e2$ **(b)** $\gamma = 1e1$ **(c)** $\gamma = 1$ **(d)** No blur

Figure 5.2: The blur over the depth of the objects decreases as γ decreases.

5.2 Optimizations

In this section, we present an application of our differentiable renderer that traditional are not capable of: optimizing scene parameters to match a reference image using gradient descent. In these experiments, a scene is rendered differentiably and the resulting image

is compared to the reference image. By calculating a scalar loss between the reference image and the rendered image, we can compute gradients for the desired scene parameters (ex.: object position, material properties) and use the gradient to compute an incremental update to the parameters to reduce the loss, which modifies the scene to more closely resemble the reference. Repeating this process, we can minimize the loss, and the images rendered and the underlying scene should match the reference. This procedure is commonly referred to as analysis-by-synthesis in inverse rendering.

We use these experiments to test if our differentiable renderer is able to properly calculate gradients with respect to different scene parameters of interest, such as the shape and position of objects and emitters, the camera pose, material properties and textures as well as some camera effects and caustics.

During optimizations, we render images with low samples per pixel (SPP) counts, which results in noisy but fast estimates. These renders are then compared to the reference image which is generated with a much higher SPP count. Optimizing based on noisy data has been shown to not be a significant problem as long as there is sufficient data [31], and we find that this holds true in our application.

We used Mean Squared Error (MSE) between the pixels of the current image and the target as our loss function for the optimization. This simple choice was effective for several optimization scenarios (vertices, specular and diffuse BRDFs, point light position). We used the Stochastic Gradient Descent (SGD) [32] optimizer available in PyTorch to update the values of the parameters.

We observe that starting with larger values for σ and γ parameters and decaying them leads to more robust optimization. This has the effect of smoothing the gradients earlier on in the optimization and can help avoid local minima.

The loss is calculated between high dynamic range (HDR) images. In scenes without visible emitters, they are lit with area lights that are explicitly sampled during the rendering.

The figures in the next few sections are organized as follows:

(a) image rendered in with initialization values,

(b) image rendered with final, optimized values,

(c) the target image used to compute the loss at every iteration,

(d) the RMS error between the initial image and the target image, and

(e) the RMS error between the final image and the target image.

5.2.1 Direct Lighting

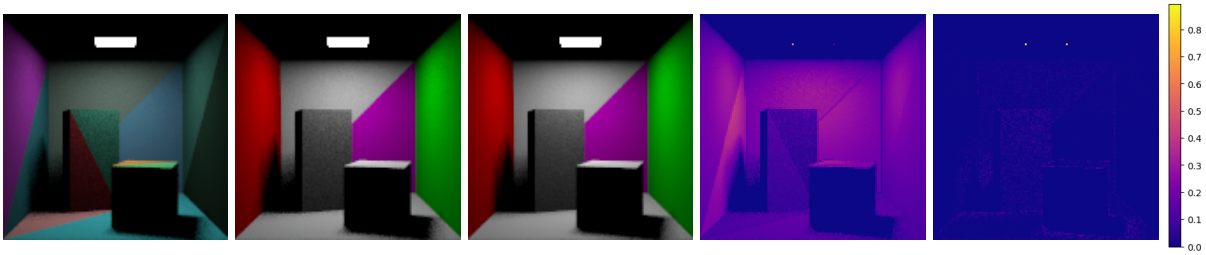
In this section, we show some examples with only direct lighting effects.

In Figure 5.3, the randomly initialized diffuse albedo of the primitives in the scene are each independently optimized to match the reference image.

In Figure 5.4, the 3D position of the camera is optimized such that the image it views matches a target image. The look at point and FOV of the camera is fixed.

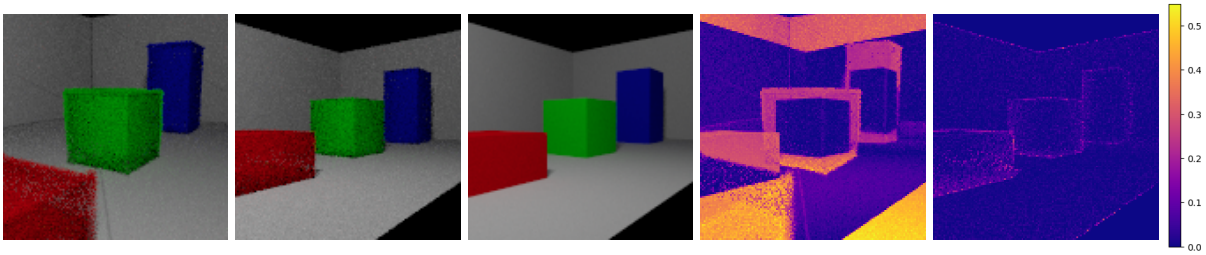
In Figure 5.5, the position of the light source is optimized to match the target image. The height of the light is fixed.

In these examples, the system can very accurately optimize the desired parameters.



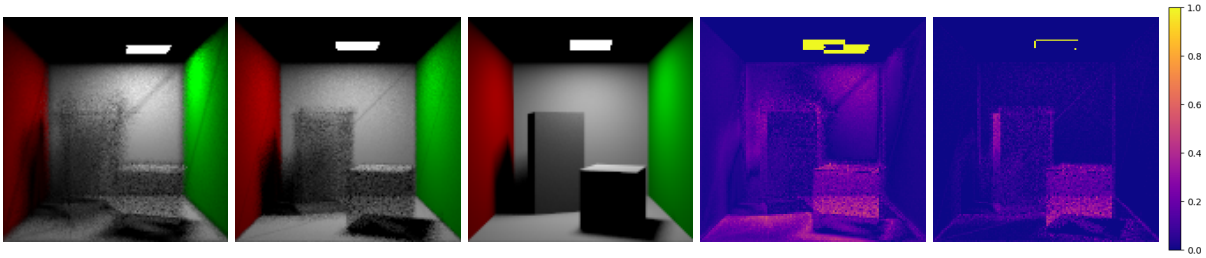
(a) Initial **(b)** Final **(c)** Target **(d)** RMSE: 0.1825 **(e)** RMSE: 0.0145

Figure 5.3: Optimize the diffuse albedo of all primitive in the scene.



(a) Initial **(b)** Final **(c)** Target **(d)** RMSE: 0.2360 **(e)** RMSE: 0.0337

Figure 5.4: Optimize the position of the camera.



(a) Initial **(b)** Final **(c)** Target **(d)** RMSE: 0.1671 **(e)** RMSE: 0.0822

Figure 5.5: Optimize the position of the light.

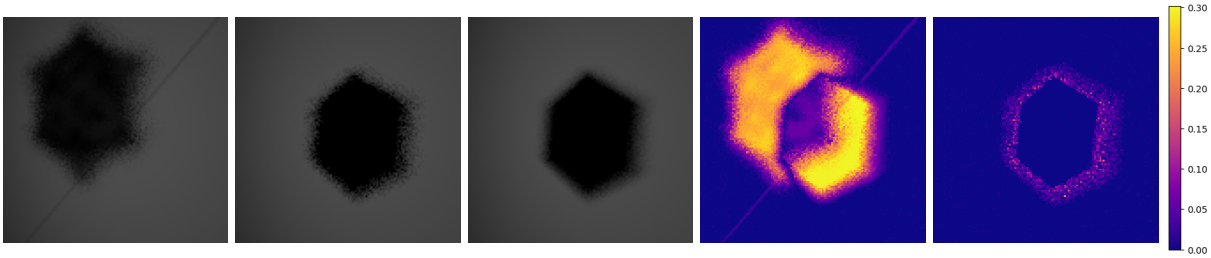
5.2.2 Higher-Order Light Transport

In this section, we present examples that demonstrate higher-order light transport effects.

In Figure 5.6, the position of an off-screen object is optimized so that its shadow matches a position given in a target image. In this setup, the only gradient signal for the translation of the object is from the shadow of the triangle. The construction of shadows in our systems allows for informative gradients to be calculated from shadows and the optimizer can easily translate the object to the correct position.

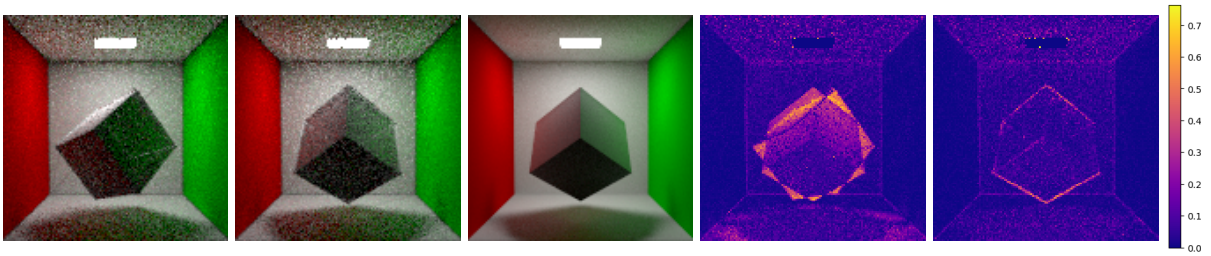
In Figure 5.7, the rotation of the cube (3 degrees of freedom) is optimized. This image features indirect light coming from the red and green walls which has a significant contribution to the appearance of the white cube. The optimizer is able to accurately rotate the cube.

In Figure 5.8, we optimize a texture on a surface that is only visible as a reflection in a highly specular / mirror-like surface. Note that the rest of the image appears black since the mirror-like surface and the textured surface are the only objects in the scene. The reconstructed texture is shown in Figure 5.9. Overall, the reconstructed texture matches the original texture quite well. However, the reconstructed texture does lack some of the high-frequency detail in the tree branches and the pattern on the tree trunk.



(a) Initial **(b)** Final **(c)** Target **(d)** RMSE: 0.1216 **(e)** RMSE: 0.0161

Figure 5.6: Optimize the position of an off screen object based on its shadow.



(a) Initial **(b)** Final **(c)** Target **(d)** RMSE: 0.1182 **(e)** RMSE: 0.0739

Figure 5.7: Optimize the rotation of the cube (3 degrees of freedom).

5.2.3 Camera Effects

In Figure 5.10 and Figure 5.11 , we present some optimizations of scenes involving motion blur and depth of field effects.

In Figure 5.10, the distance to the focus point of the camera is optimized. The initial focal point falls roughly in the plane of the blue box, which causes it to appear most in

(a) Initial **(b)** Final **(c)** Target **(d)** RMSE: 0.0525 **(e)** RMSE: 0.0107

Figure 5.8: Optimize the texture on a surface that is only visible in a mirror-like surface.

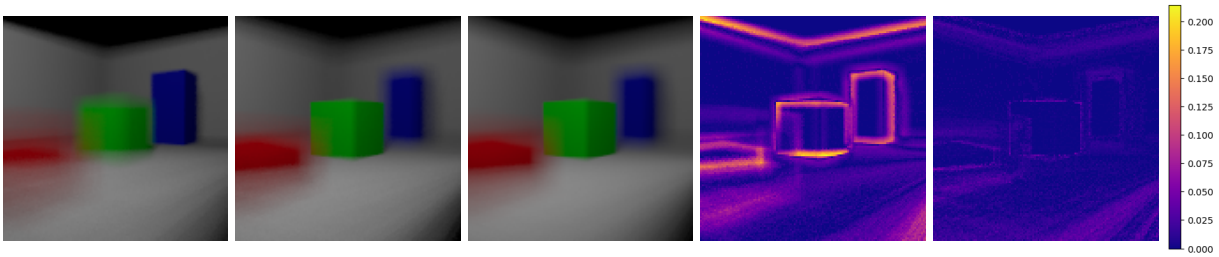
(a) Target **(b)** Final

Figure 5.9: Target texture and the texture recovered during the optimization from Figure 5.8. We can see that the general shape of the tree is recovered, but the high-frequency checkerboard pattern is not.

focus. The optimizer is correctly able to adjust the focal distance so that the green box is properly in focus.

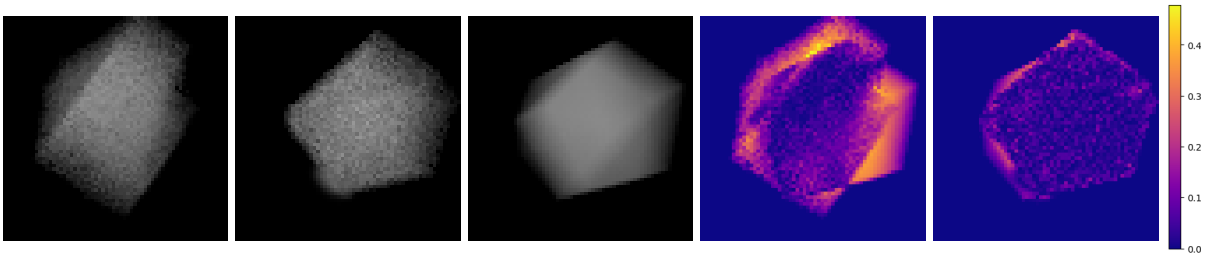
In Figure 5.11, the velocity and the geometry (vertex positions) of the object are optimized simultaneously, from a single viewpoint. Each vertex in the object is randomly perturbed independently. This situation is quite challenging, even when the motion is constrained to be linear. In the final image with motion blur, the appearance of the object is improved over the initial image, but there is still some error around the edges. The initial, final, and target images are rendered without motion blur in Figure 5.12. We can

see in the final image without motion blur that the final shape is better than the initial and has captured some key features of the target, but it lacks detail particularly in the areas where faces were only partially visible.



(a) Initial **(b)** Final **(c)** Target **(d)** RMSE: 0.0455 **(e)** RMSE: 0.0116

Figure 5.10: Optimize the distance of the focal point of the camera to match the depth of field effect.

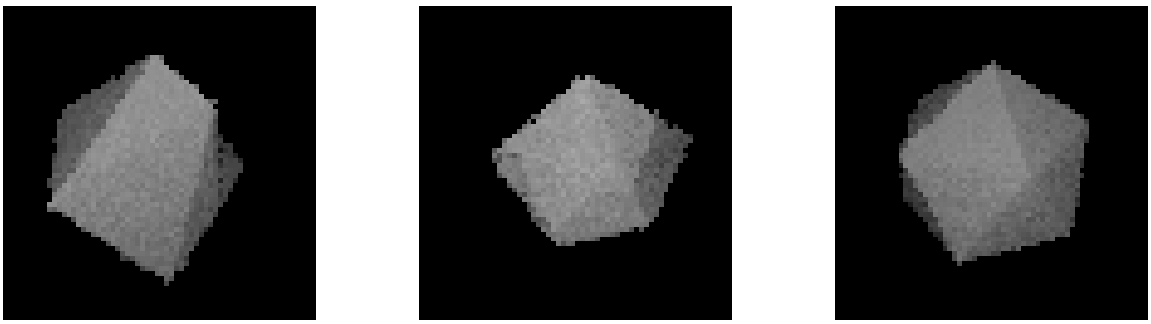


(a) Initial **(b)** Final **(c)** Target **(d)** RMSE: 0.1118 **(e)** RMSE: .0431

Figure 5.11: Optimize to determine the vertices and velocity of the object in an image with significant motion blur.

5.2.4 Caustics

For the optimization of the caustics implemented in our system, we found that optimizations using MSE loss struggled. For these cases, we investigated some other loss functions. A loss function that we found to have an improvement over MSE was the symmetric mean absolute percentage (SMAPE) error function. This loss function is supposed to have stable behaviour with HDR images [33], and we found this to be true in our application.



(a) Initial **(b)** Final **(c)** Target

Figure 5.12: Snapshots from the motion blur geometry optimization, with no motion blur. Not used in the optimization process.

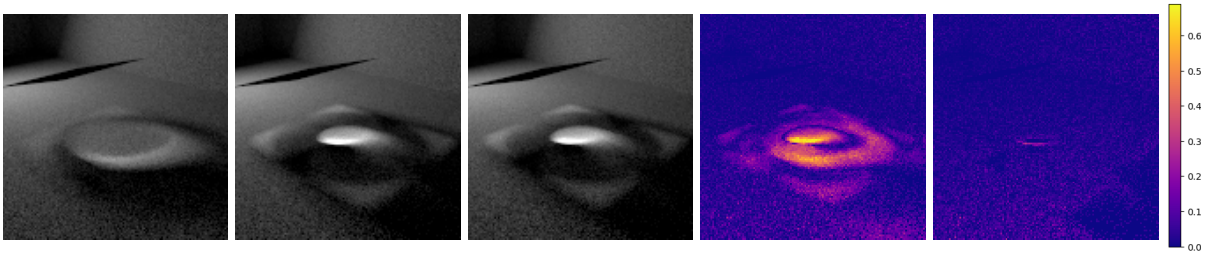
These examples use only light tracing. As a result, the glass pane appears black in the image, as the specular surface cannot form a direct connection to the camera. Light tracing is necessary to efficiently render caustics when using a point camera.

In Figure 5.13, the index of refraction (IOR) of a glass sheet with a normal map that creates a lensing effect is optimized. There is only a single parameter for the whole glass pane. The system can easily optimize the parameter.

In Figure 5.14, the position of the caustic optimized. Most of the gradient signal in this example comes from the caustic itself, rather than from the glass pane that is only visible from a very sharp angle. This is demonstrated in Figure 5.15. We are able to reposition the glass pane so that the caustic matches the target image more closely, but the optimization gets caught in a local minimum and is not able to perfectly reproduce the target image. This situation appears to be difficult since there is no significant contrast between the area of the caustic and the diffuse shading on the surface.

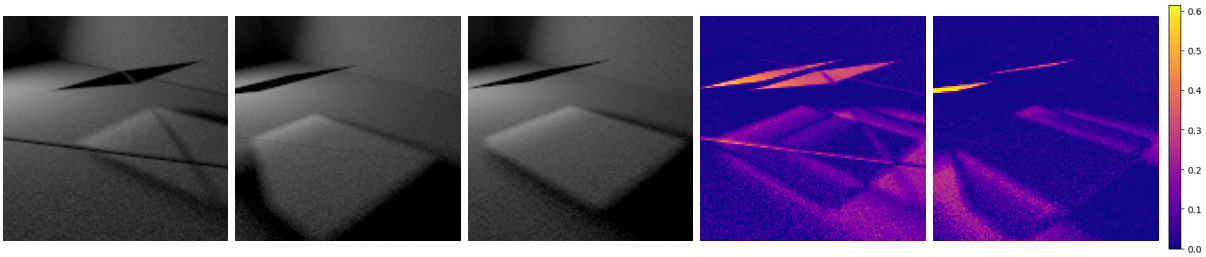
We experimented with smoothing out the edges of the caustic by sampling a Phong lobe around the refraction direction, but this did not result in a significant improvement.

We also tried this example with the caustic produced with the normal map as in Figure 5.13, but in that case, the system was not able to translate the glass pane to the correct position at all.



(a) Initial **(b)** Final **(c)** Target **(d)** RMSE: 0.1374 **(e)** RMSE: 0.0486

Figure 5.13: Optimize the index of refraction of a sheet of glass with a normal map that focuses the light to produce a lens effect.



(a) Initial **(b)** Final **(c)** Target **(d)** RMSE: 0.0999 **(e)** RMSE: 0.0738

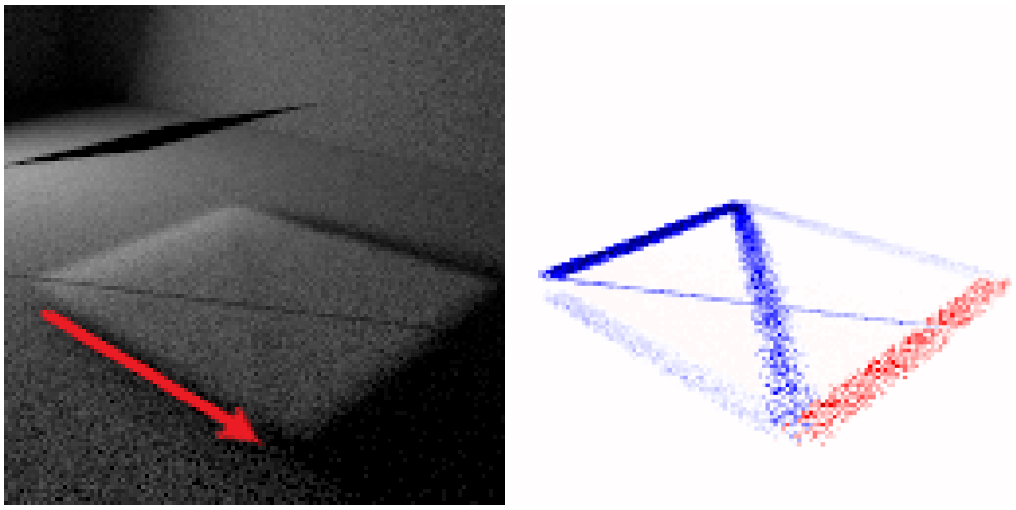
Figure 5.14: Optimize the position of the caustic.

5.3 Renderer Gradients

In Figure 5.16, we render a torus with our differentiable rendering method with different values of σ and visualize the corresponding gradients for a translation in the x-direction (right positive, left negative). The reference image shows the object with no blurring. In the heat map images of the gradients, red denotes an increase in the average RGB value of the pixel (i.e. moving from black to red in this case), while blue denotes a decrease.

In the $\sigma = 1\text{e}{-4}$ image, while the rendered image of the torus is quite blurring, the gradient heat map paints a clear picture of which pixels would increase or decrease in colour as the torus moves to the right.

We can see as we decrease the value of σ, the rendered image more closely resembles the reference image. The image with $\sigma = 1\text{e}{-6}$ is almost identical to the reference, apart



(a) (b)

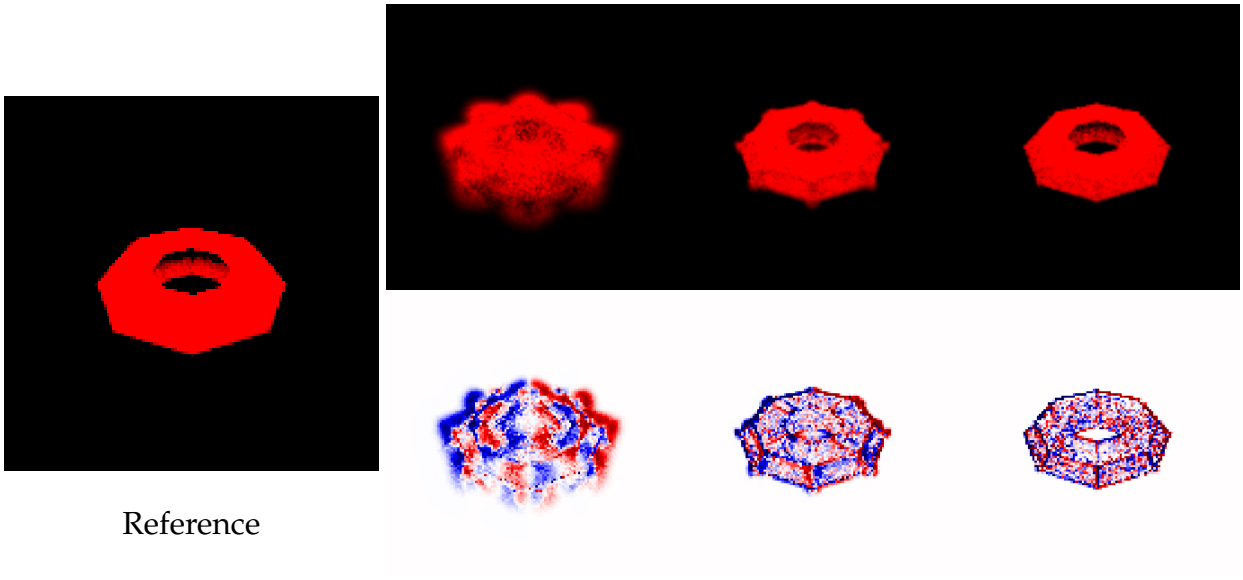
Figure 5.15: An image with a caustic from Figure 5.14. The gradients for a translation of the glass pane in +x direction (red arrow). The gradients for the translation of the plane appear in the area of the caustic.

from the noise that is a result of using fewer SPP for generating the gradient images. However, as σ decreases the gradients do appear to become noisier.

5.4 Comparison with REDNER

In Figure 5.17, we show an example where our system can successfully optimize the placement of an object, while the REDNER [8] fails. In this example, in which only the diffuse albedo of surfaces directly visible by the camera, the torus is initially displaced away from the center of the image. The initial position of the torus is such that none of the pixels of the initial image and the target image overlap, and part of the object is off-screen.

In these cases, the gradients calculated by REDNER cause the optimizer to shift the object off-screen so that the pixels that are currently covered by the object become black which is a local minimum in the loss function. Part of the object being off-camera is



Reference

(a) $\sigma = 1e{-}4$ **(b)** $\sigma = 1e{-}5$ **(c)** $\sigma = 1e{-}6$

Figure 5.16: Pixel gradients for translation in the x-direction (from left to right) with different values of σ. Image is rendered with 16 SPP and direct lighting only.

important for this to happen. If we increase the field of view such that the entire off-center image, REDNER is again able to center the object correctly, as we show in Figure 5.18.

When part of the object is not visible in the image, the gradients for it are not calculated. We think that this causes an imbalance in the total gradient for the translation of the torus, which drives it further off-screen. The missing pixels do not contribute gradient information that moving the object further away from the center is not actually an improvement.

In our system, we can successfully optimize the position of the torus when it is partially off-screen. It does struggle somewhat, as the torus does not seem to take a direct path to the center of the image, instead moving up and right, before finding the center.

The regularization introduced by our methods widens the footprint of the torus in the early images of the optimization so that it overlaps partially with the position of the object

in the target image. We think that this has the effect of smoothing out the loss landscape, making it more convex and easier for the optimizer to find a path to the global minimum. As the optimization progresses, the regularization in the image is decreased, and the optimizer can correctly fine-tune the position after the coarse initial updates brought it to approximately the correct position.

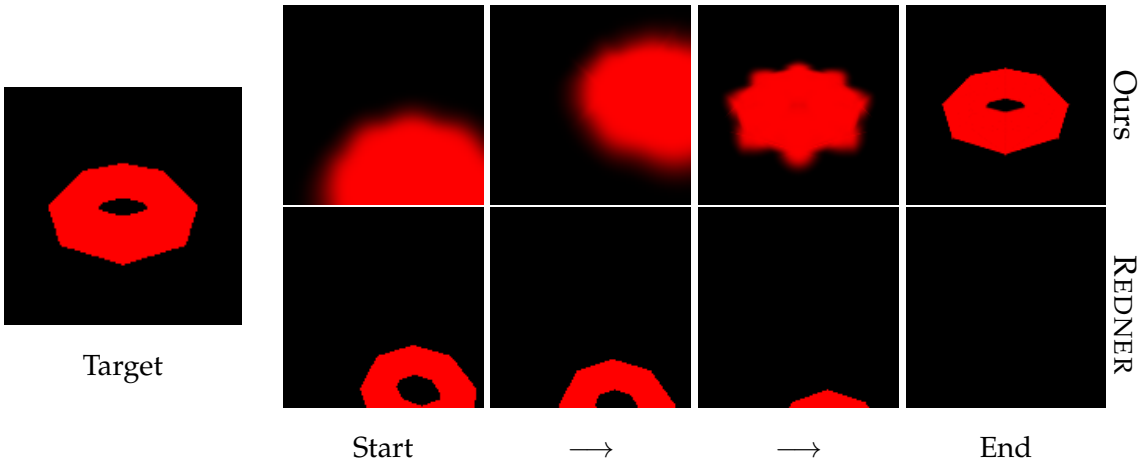


Figure 5.17: Ours vs REDNER for the translation of a torus. The image of the torus in its initial position does not overlap with its target, and the torus is partially off-screen. With this setup, REDNER fails to position the torus correctly. Our system introduces blur, which smooths the loss and allows for the correct position to be found with this adverse initialization. The images between 'Start' and 'End' are taken at illustrative points of the optimization.

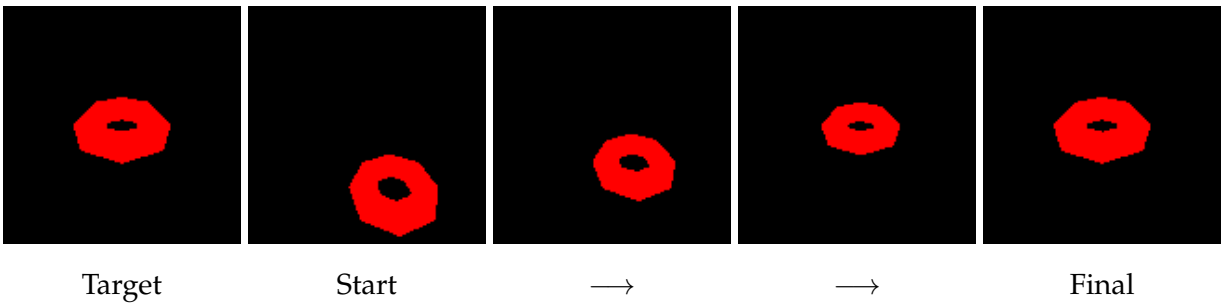


Figure 5.18: Another translation of a torus in REDNER. In this example, the FOV is wider than in Figure 5.17 and the whole object is visible in the initial position. REDNER can correctly position the object to match the target.

	SoftRas	DIB-R	Redner	Mitsuba 2	**Ours**
Single Objects	✔	✔	✔	✔	✔
Full Scenes	✖	✖	✔	✔	✔
GI	✖	✖	✔	✔	✔
Textures	✗	✔	✔	✔	✔
Emitters	✖	✗	✔	✔	✔
Camera Effects	✗	✖	✖	✖	✔
Caustics	✖	✖	✖	✔	✗
Performace	✗	✗	✖	✖	✗
Scalability	✖	✖	✔	✔	✗
Flexibility	✖	✖	✔	✔	✔
Occlusion	✔	✗	✖	✖	✔
Gradients	✗	✗	✔	✗	✗
Noise-free	✔	✔	✖	✖	✖
Tuning	✔	✔	✖	✗	✔

Table 5.1: Comparison of characteristics of some differentiable rendering systems. *Single Objects* denotes if the differentiable system can calculate gradients for an image of a unique object. *Full Scenes* denotes if the system can calculate gradients for scenes containing multiple objects. *GI* refers to if the system supports global illumination. *Textures* indicates if the method supports textures. *Emitters* denotes if the system can calculate gradients for light sources. *Camera Effects* and *Caustics* indicate if the system supports these features respectively. *Performance* considers capability for rapid rendering in applications such as in deep learning. *Scalability* refers to performance with larger numbers of primitives and higher image resolution. *Flexibility* is whether the system is designed to support arbitrary shading. *Occlusion* denotes if the method can compute gradients for occluded surfaces. *Gradients* refers to the correctness/unbiasedness of gradients. *Noise-free* systems do not rely on random sampling. *Tuning* refers to tunable parameters that affect the rendered image or gradients.

Chapter 6

Discussion

6.1 Benefits

One advantage of our approach compared to other physically based differentiable rendering systems is that we do require tracing extra auxiliary rays to compute gradients, nor do we need any sort of data structure to keep track of the edges/silhouettes of objects. In REDNER and MITSUBA 2, these steps are required and add a large amount of overhead computation to calculate the gradients.

A similar idea of blurring scene geometry to remove discontinuities was investigated by Rhodin et al. [34]. Their method involves first fitting the geometry first to a number of spheres and then replacing the spheres with Gaussian blurs. They did not show global illumination, even though their method seems capable of support it.

6.1.1 Implementation

Implementing this system using Python and PYTORCH worked well. While, other differentiable rendering methods typically have dependencies on C++ libraries, which can

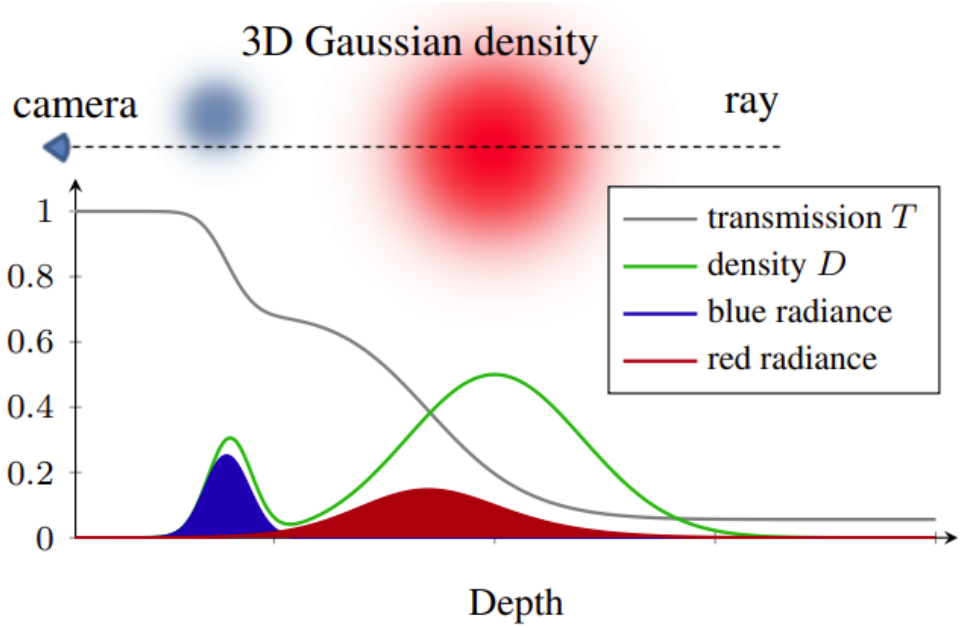


Figure 6.1: Rhodin et al. [34] approximate geometry with Gaussian blurs that emit light. The pixel color is the fraction of source radiance that is emitted along the ray and reaches the camera. The top of the figure shows a ray leaving the camera and passing through two 3D Gaussian densities. The bottom shows the light being transport along the ray. The density along the ray is a sum of 1D Gaussians (green). The transmittance (gray) decays from one as we move away from the camera. The fraction of reflected light that reaches the camera (red and blue areas) is the radiance used to compute the visibility. *Source: Rhodin et al. [34]*

often be difficult to configure properly on a variety of systems, Python and Python libraries are extremely portable and system agnostic.

Using PyTorch allowed us to easily leverage GPUs to speed up the rendering operations. PyTorch also includes built-in automatic differentiation.

Another advantage of using the PyTorch library is the ease of integration into common machine learning pipelines. PyTorch is one of the most popular machine learning libraries. We hope that the system we developed and the code we plan to make available

will find uses in the development and training of unsupervised models. Their apply their method to pose estimation problems.

6.1.2 Camera Effects

We show some examples of our system being applied to camera effects. To our knowledge, there have not been any papers published that explore this application. We find that performing inverse rendering tasks on images with depth of field effects and motion blur are relatively easy to implement. Both of these effects involve making multiple rendering passes and then averaging the images together. This averaging operation is easily handled by automatic differentiation, so no special consideration has to be made. The main complication that arises as a result of doing multiple passes is that the memory requirements can become large since the derivative information for each pass must be stored.

6.2 Limitations

While using PyTorch was very convenient and easy, it may also introduce some limitations. PyTorch is optimized for operations done most commonly in machine learning, like matrix multiplications. Our system uses a wide variety of operations, some of which may not be as heavily optimized. It is possible that writing a custom library or custom C++/CUDA functions focused on our use case might improve the efficiency of the renderer. It also may be possible to include some analytic or compact derivative calculations at certain parts of the rendering process, which could further increase the speed. One use case where this reliance on PyTorch is a significant drawback is for the rendering of images of the gradients, as in Section 5.3. To produce these images, we compute the gradient for each pixel, which requires a backwards pass through the computation graph

for each pixel in the image. There is no way to do this in a batched way utilizing the GPU in PyTorch, so the process is extremely slow.

One of the biggest drawbacks of our system, as it is currently implemented, is that it scales poorly with more geometry. As the geometry in the scene becomes more complex (i.e. more triangles) we have to test for the intersection with each of these triangles.

In traditional renderers, this is often handled by implementing an acceleration data structure. One such example is a Bounding Volume Hierarchy (BVH), which partitions the triangles in the scene spatially in such a way that they can be tested for intersections as a group, reducing the computation needed from $\mathcal{O}(N)$ to $\mathcal{O}(\log N)$.

Due to the way our algorithm blurs the edges of geometry outside the enclosed area of the triangle, a traditional BVH type structure would not work, as it would these areas. It may be possible to construct some type of BVH where padding is added around the triangles when constructing the BVH and we accept the clipping away of areas that are sufficiently far from the edge of a triangle and would have $\alpha \sim 0$ anyways.

Another issue with using a BVH-type structure would arise when optimizing the position or shape of objects in a scene. A BVH is useful when a scene is static; if things in the scene move around significantly, the BVH will no longer be accurate and useful. This could possibly be overcome without too significant of a decrease in performance by loosely fitting the BVH to the scene, but if the geometry shifted beyond the padding, problems would arise. An alternative would be to rebuild the BVH, either at every iteration of the optimization or periodically, to ensure its accuracy. This option is not very attractive, as building a BVH can take significant time.

Our depth weighting term can cause areas far from the camera to appear black if γ is set to too small of a value $\sim \gamma < 1$.

6.3 Differentiable Renders in General

In this section, we discuss some of the concerns and open questions related to the calculation of gradients from a rendered image and the applications that make use of these gradients.

6.3.1 Exact Gradients and Bias

In recent differentiable work, there has been discussion of the importance of producing unbiased gradients.

In the work by Bangaru et al. [21] on "Unbiased warped-area sampling for differentiable rendering", they present a method similar to Loubet et al. [10] for computing gradients. A core component of the Bangaru et al. work is that the method they develop can produce unbiased estimates of the gradients, while the Loubet et al. method is biased. The bias in the Loubet et al. work is due to some approximations and assumptions they make to simplify their computations. Bangaru et al. show that that the Loubet et al. method is actually a special case of their method, that neglects to satisfy some conditions that would be required for unbiased gradients.

While Bangaru et al. put significant emphasis on their unbiased estimators, in many of their examples they actually use the consistent estimator they developed, rather than the unbiased one, as it is faster and produces good results. This brings up the question of whether unbiased gradients are truly necessary or even desirable. A common theme in calculations that involving random sampling is that there is a trade-off between bias and variance; reducing one usually means increasing the other.

In general, it is usually desirable to have unbiased gradients for stochastic gradient descent, if these gradients are very noisy, they can be uninformative and the optimization will have difficulty converging. In this case, it is better to have accurate and low-variance gradients that allow a local minimum to be found in fewer iterations. However, once in

the vicinity of the local minimum, the bias could potentially adversely affect the convergence during the last steps of optimization.

Our differentiable rendering algorithm does not produce exact gradients and introduces bias into the image. However, like prior approaches [5, 6, 7], we find that during optimization it is often not necessary to produce *exact* gradients, but rather ones that are useful for finding a minimum.

6.3.2 Inclusion of Occlusion Discontinuity Differentiability

We consider surfaces that are technically out of view of the camera, as they are occluded by other objects, similar to differentiable rendering approaches taken in rasterization-based methods like SOFTRAS [5].

On the other hand, the physically-based differentiable renderers like REDNER [8] and MITSUBA 2 [9] specifically disregard occluded surfaces entirely.

Considering occluded surfaces introduces some complications. Typically, the surfaces need to have some sort of transparency built-in, that allows for occluded objects to still have a contribution to some pixels in the output image, and therefore gradients can be propagated to the parameters of the object. The inclusion of this transparency can be tricky, as it introduces bias to the image. It also complicates the computation of shadows, and refractive surfaces as well.

While there are these complications to consider, there are also some advantages to calculating gradients for occluded objects. For example, it can allow for the optimization of the placement of objects in some scenarios. In SOFTRAS [5], they show a nice example of a model of a human and show that they can properly reposition one of the hands from the far side of the body (that would normally be out of view) to a correctly oriented pose in front of the body. Applications such as this seem to make the inclusion of gradient

calculation for some occluded objects compelling, and it would be interesting to more physically based methods that included this.

The depth blurring function we use is similar to that used in Order Independent Transparency [35] by McGuire and Bavoil. The McGuire and Bavoil [35] work also discusses other functions for weighting transparent surfaces, that may be worth exploring more.

6.3.3 Initialization

In differentiable rendering applications where we want to optimize something in a scene, we first require some initial description of the scene. Sometimes obtaining this initial description can be difficult, for example in cases where are dealing with real-world photographs. Fortunately, advances in computer vision are making it possible to obtain decent initialization. However, if the initial scene description is very inaccurate, any step involving differentiable rendering will likely fail.

6.3.4 Self Intersections

In systems like Mitsuba 2 [10, 9], the authors state that their method for handling edge discontinuities relies on there being only one type of discontinuity, at the edges of silhouettes of objects. The system outlined in "Differentiable Path Space Rendering" [12], states similar assumptions.

One situation that may arise when using differentiable rendering for the optimization of scene geometry is that another source of discontinuities may be introduced: intersections of geometry. Geometry intersections could arise in the case of trying to position an object in a scene. For example, trying to position a cup on a table. For some iterations of the optimizations, the cup could be translated in a way that it intersects with the surface of the table.

Additionally, another situation in which geometry intersection could arise would be in cases where we are trying to optimize the shape of an object. If the displacements of the vertices in the mesh being optimized are not constrained in some way, it could be possible for the vertices to be moved in such a way that the mesh now has self-intersections.

6.4 Implicit Representations in Differentiable Rendering

In rendering, most commonly we deal with explicit representations of objects or scenes. Some examples of common explicit representations are meshes, voxel grids, and point clouds. In these representations, the surface, shape or volume is described by exhaustively listing out every single triangle, voxel or point we want to be included in the image. The memory usage of these types of representations scales proportionally to the complexity or detail of whatever we are trying to represent. While these representations are often convenient and useful, they have some inherent difficulties when it comes to incorporating them into some machine learning applications. They can have difficulties representing curved surfaces, and the topology (connectivity) can be fixed, which can cause difficulties when we wish to modify the mesh with an automated system.

An implicit representation is a way of representing a scene or an object without directly listing out a series of primitives. For example, if we wanted to render a sphere, we could solve for the intersection between a ray $R(O, d) = O + td,\ t \geq 0$ and the equation describing a sphere with radius r centered at the origin $x^2 + y^2 + z^2 = r^2$. By using an equation to describe the sphere, we avoid having to tessellate the sphere with square or triangular primitives, which would introduce some approximation error as these flat primitives cannot perfectly capture the curvature of the sphere. Storing the parameters that describe the sphere (only the radius in this case) is also more compact than, for example, storing a list of the many faces and vertices that would be required to give a sufficiently smooth-looking sphere.

These representations have been investigated more in recent years, with multi-layer perceptron (MLP) and other neural network models, being used to store the data that represents the scene or object. Implicit representations have several advantages. Implicit representations can be more compact in terms of memory usage when representing highly detailed structures, and as they represent scenes in a continuous manner, they can be more amenable to machine learning systems.

One of the current downsides of implicit representations is that they typically have to be converted to explicit representations, to work with standard rendering systems. This step can introduce significant overhead in rendering time.

In 2020, Neural Radiance Fields (NeRF) [36] appeared. In this work, the authors trained an MLP to output the incoming radiance to give a point and a direction. They did this by optimizing the MLP to fit a digital scene with global light transport effects viewed from multiple camera angles, and the MLP was able to effectively generalize what the scene would like from novel camera angles. This work produced some delightful scenes where the camera could be moved around and the scene could be rendered from the novel positions with complex light transport effects simply by querying the MLP.

This work spawned many follow-ups in short order, such as D-NeRF [37] which can produce so-called "nerfies" of faces from cellphone captured video, and DONeRF [38] which adds a time dimension allowing for novel viewpoints to be generated in a video.

Other implicit representations, such as Neural Signed Distance Functions (SDF), have also become seen increased interest and success [39]. These representations are typically an MLP or NN that can be queried with a point in space, and they return the distance to the surface of the scene or object, with a positive sign for outside and a negative sign if the point lies inside. By determining the zero level set of the function, the surface of the object can be recovered. The advantage here is that these representations can capture large or detailed scenes with a reduced memory footprint.

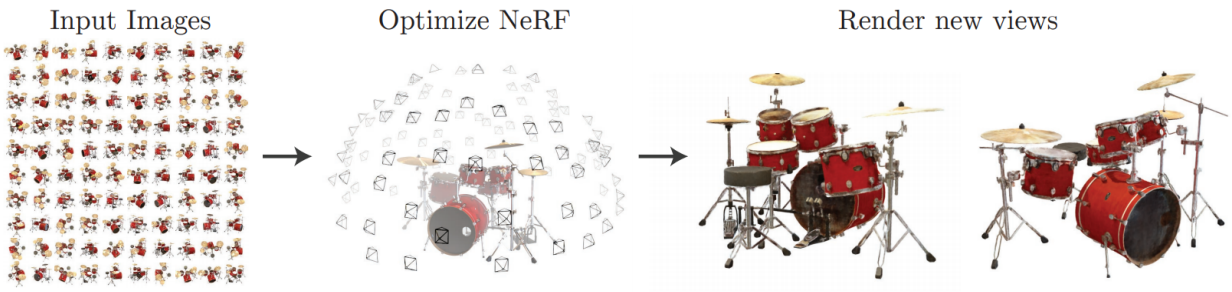


Figure 6.2: The NeRF [36] method fits a continuous 5D representation (volume density and view-dependent colour at any continuous location) of a scene from a set of input images. A technique similar to raymarching is used to accumulate samples from the scene representation along rays to render the scene from any viewpoint. Here, a set of 100 input views of a synthetic drum kit scene are randomly captured on a surrounding hemisphere and used to generate the NeRF. Then, two novel views were rendered using the NeRF. *Source: Mildenhall et al. [36]*

Utilizing implicit representations in differentiable rendering may help to overcome difficulties in reconstructing/representing geometry. One of the biggest issues when trying to optimize the geometry of a mesh is that the topology of the mesh is fixed. This creates problems where the genus of the target does not match the genus of the starting mesh. For example, if we start with a sphere mesh (genus 0, no holes) and want to transform it into a donut or a coffee cup with a handle (both genus 1 shapes, 1 hole), we would require changing topology (connectivity) of the vertices in the mesh to add a hole. Figuring out exactly how to change the topology to achieve the desired target is very challenging, and is something that is currently not supported in differentiable renders or ML models that integrate differentiable renders.

Chapter 7

Conclusion

We presented our novel method for differentiable rendering with global illumination. Our method is based on the idea of regularizing the discontinuities that appear in the rendering of images, to make the rendering function differentiable. While this introduces some bias into the rendered image, it admits the computation of useful gradients without the need for tracing additional rays or data structures for storing edge information.

We showed how our method can be used in a variety of inverse rendering cases, optimizing material properties, camera placement, and object positions and geometry. We also show how camera effects such as motion blur and depth of field can be easily integrated into differentiable renderer, and how parameters related to these effects can be automatically determined. We also explore some simplified differentiable caustics.

While our system has some limitations when it comes to the speed and number of primitives it could efficiently handle, the underlying theory of the approach is simple and effective.

Differentiable rendering, particularly physically-based differentiable rendering, is a field that has recently seen significant progress and interest. It will be interesting to see how the field will progress in the coming years. Something important to the continued

interest and advancement in this field will likely be finding an application where the complex light transport that physically based rendering can handle allows for improvements, over the faster rasterization methods. At the time of the writing of this thesis, most of the physically based differentiable renderers are limited to working on synthetic data

Hopefully, physically-based differentiable rendering will find its place in planning the placement of light sources or windows in architectural simulations, analyzing weather satellite images based on volumetric scattering in cloud formations, helping to bridge the sim2real gap in reinforcement learning, holography, or in some other clever way.

7.1 Differentiable Rendering – Perspectives and Future Work

There are still some effects that are common in rendering that have not appeared in differentiable rendering, such as subsurface scattering. Subsurface scattering happens when light can penetrate a shallow distance into a material before bouncing out. This effect contributes strongly to the appearance of skin (particularly paler skin tones) and coloured liquids such as milk.

Facilitating the ease of integration of differentiable rendering into high-level applications and machine learning seems to be a priority for everyone working in the field. REDNER became available as a pip package with python functions for all its features and MITSUBA 2 released with one of its core features being fine-grained python bindings. General rasterization-based packages like NVDIFFRAST [40] and PYTORCH3D [29] have also appeared over the last couple of years.

Bibliography

[1] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[2] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[3] Hiroharu Kato, Deniz Beker, Mihai Morariu, Takahiro Ando, Toru Matsuoka, Wadim Kehl, and Adrien Gaidon. Differentiable rendering: A survey. *arXiv preprint arXiv:2006.12057*, 2020.

[4] Matthew M. Loper and Michael J. Black. Opendr: An approximate differentiable renderer. In *European Conference on Computer Vision*, pages 154–169. Springer, 2014.

[5] Shichen Liu, Tianye Li, Weikai Chen, and Hao Li. Soft rasterizer: A differentiable renderer for image-based 3d reasoning. *The IEEE International Conference on Computer Vision (ICCV)*, Oct 2019.

[6] Wenzheng Chen, Huan Ling, Jun Gao, Edward Smith, Jaakko Lehtinen, Alec Jacobson, and Sanja Fidler. Learning to predict 3d objects with an interpolation-based differentiable renderer. In *Advances in Neural Information Processing Systems*, pages 9609–9619, 2019.

[7] Hiroharu Kato, Yoshitaka Ushiku, and Tatsuya Harada. Neural 3d mesh renderer, 2017.

[8] Tzu-Mao Li, Miika Aittala, Frédo Durand, and Jaakko Lehtinen. Differentiable monte carlo ray tracing through edge sampling. *ACM Trans. Graph. (Proc. SIGGRAPH Asia)*, 37(6):222:1–222:11, 2018.

[9] Merlin Nimier-David, Delio Vicini, Tizian Zeltner, and Wenzel Jakob. Mitsuba 2: A retargetable forward and inverse renderer. *Transactions on Graphics (Proceedings of SIGGRAPH Asia)*, 38(6), November 2019. doi: 10.1145/3355089.3356498.

[10] Guillaume Loubet, Nicolas Holzschuch, and Wenzel Jakob. Reparameterizing discontinuous integrands for differentiable rendering. *ACM Transactions on Graphics*, December 2019.

[11] Merlin Nimier-David, Sébastien Speierer, Benoît Ruiz, and Wenzel Jakob. Radiative backpropagation: an adjoint method for lightning-fast differentiable rendering. *ACM Transactions on Graphics (TOG)*, 39(4):146–1, 2020.

[12] Cheng Zhang, Bailey Miller, Kai Yan, Ioannis Gkioulekas, and Shuang Zhao. Path-space differentiable rendering. *ACM Transactions on Graphics (TOG)*, 39(4):143–1, 2020.

[13] Shuang Zhao, Wenzel Jakob, and Tzu-Mao Li. Physics-based differentiable rendering: From theory to implementation. In *ACM SIGGRAPH 2020 Courses*, SIGGRAPH 2020, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450379724. doi: 10.1145/3388769.3407454. URL `https://doi.org/10.1145/3388769.3407454`.

[14] James T. Kajiya. The rendering equation. *SIGGRAPH Comput. Graph.*, 20(4):143–150, August 1986. ISSN 0097-8930. doi: 10.1145/15886.15902. URL `https://doi.org/10.1145/15886.15902`.

[15] J Carlson. Monte carlo methods and applications in nuclear physics. Technical report, Los Alamos National Lab., 1990.

[16] Martin H. Weik. *Snell's law*, pages 1607–1607. Springer US, Boston, MA, 2001. ISBN 978-1-4020-0613-5. doi: 10.1007/1-4020-0613-6_17633. URL https://doi.org/10.1007/1-4020-0613-6_17633.

[17] Matt Pharr, Wenzel Jakob, and Greg Humphreys. *Physically Based Rendering: From Theory to Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3rd edition, 2016. ISBN 0128006455.

[18] Eric Veach and Leonidas Guibas. Bidirectional estimators for light transport. In Georgios Sakas, Stefan Müller, and Peter Shirley, editors, *Photorealistic Rendering Techniques*, pages 145–167, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg. ISBN 978-3-642-87825-1.

[19] Henrik Wann Jensen. *Realistic image synthesis using photon mapping*, volume 364. Ak Peters Natick, 2001.

[20] Toshiya Hachisuka, Shinji Ogaki, and Henrik Wann Jensen. Progressive photon mapping. In *ACM SIGGRAPH Asia 2008 papers*, pages 1–8. 2008.

[21] Sai Bangaru, Tzu-Mao Li, and Frédo Durand. Unbiased warped-area sampling for differentiable rendering. *ACM Trans. Graph.*, 39(6):245:1–245:18, 2020.

[22] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.

[23] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library, 2019.

[24] Wenzel Jakob. Enoki: structured vectorization and differentiation on modern processor architectures, 2019. https://github.com/mitsuba-renderer/enoki.

[25] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017.

[26] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks, 2014.

[27] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples, 2015.

[28] Peter W Glynn and Donald L Iglehart. Importance sampling for stochastic simulations. *Management science*, 35(11):1367–1392, 1989.

[29] Nikhila Ravi, Jeremy Reizenstein, David Novotny, Taylor Gordon, Wan-Yen Lo, Justin Johnson, and Georgia Gkioxari. Accelerating 3d deep learning with pytorch3d. *arXiv:2007.08501*, 2020.

[30] Krishna Murthy J., Edward Smith, Jean-Francois Lafleche, Clement Fuji Tsang, Artem Rozantsev, Wenzheng Chen, Tommy Xiang, Rev Lebaredian, and Sanja Fidler. Kaolin: A pytorch library for accelerating 3d deep learning research. *arXiv:1911.05063*, 2019.

[31] Jaakko Lehtinen, Jacob Munkberg, Jon Hasselgren, Samuli Laine, Tero Karras, Miika Aittala, and Timo Aila. Noise2noise: Learning image restoration without clean data. *arXiv preprint arXiv:1803.04189*, 2018.

[32] Sebastian Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016.

[33] Thijs Vogels, Fabrice Rousselle, Brian McWilliams, Gerhard Röthlin, Alex Harvill, David Adler, Mark Meyer, and Jan Novák. Denoising with kernel prediction and asymmetric loss functions. *ACM Transactions on Graphics (TOG)*, 37(4):1–15, 2018.

[34] Helge Rhodin, Nadia Robertini, Christian Richardt, Hans-Peter Seidel, and Christian Theobalt. A versatile scene model with differentiable visibility applied to generative

pose estimation. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 765–773, 2015.

[35] Morgan McGuire and Louis Bavoil. Weighted blended order-independent transparency. *Journal of Computer Graphics Techniques (JCGT)*, 2(2):122–141, December 2013. ISSN 2331-7418. URL `http://jcgt.org/published/0002/02/09/`.

[36] Ben Mildenhall, Pratul P. Srinivasan, Matthew Tancik, Jonathan T. Barron, Ravi Ramamoorthi, and Ren Ng. Nerf: Representing scenes as neural radiance fields for view synthesis. In *ECCV*, 2020.

[37] Keunhong Park, Utkarsh Sinha, Jonathan T. Barron, Sofien Bouaziz, Dan B Goldman, Steven M. Seitz, and Ricardo Martin-Brualla. Deformable neural radiance fields. *arXiv preprint arXiv:2011.12948*, 2020.

[38] Thomas Neff, Pascal Stadlbauer, Mathias Parger, Andreas Kurz, Chakravarty R. Alla Chaitanya, Anton Kaplanyan, and Markus Steinberger. Donerf: Towards real-time rendering of neural radiance fields using depth oracle networks, 2021.

[39] Towaki Takikawa, Joey Litalien, Kangxue Yin, Karsten Kreis, Charles Loop, Derek Nowrouzezahrai, Alec Jacobson, Morgan McGuire, and Sanja Fidler. Neural geometric level of detail: Real-time rendering with implicit 3D shapes. *arXiv preprint arXiv:2101.10994*, 2021.

[40] Samuli Laine, Janne Hellsten, Tero Karras, Yeongho Seol, Jaakko Lehtinen, and Timo Aila. Modular primitives for high-performance differentiable rendering. *ACM Transactions on Graphics*, 39(6), 2020.